

USENIX Association

Dallas

1985 Winter Conference Proceedings

USENIX

Winter Conference  
Dallas 1985  
PROCEEDINGS



**USENIX Association**

**Winter Conference**

**Dallas 1985**

**PROCEEDINGS**

**January 23-25, 1985  
Dallas, Texas, USA**

For additional copies of these proceedings, write:

USENIX Association

P.O. Box 7

El Cerrito, CA 94530 USA

Price \$20.00 plus \$15.00 for overseas airmail

(c) copyright 1985 by The USENIX Association

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain  
with the author or the author's employer.

UNIX is a trademark of AT&T

Other trademarks noted in the text.

# ACKNOWLEDGEMENTS

Sponsored by:

USENIX Association  
P.O. Box 7  
El Cerrito, CA 94350

Program Co-Chairpersons:

Charisse Castagnoli   Teknekron Infoswitch  
Rob Kolstad                Convex Computer Corp.

Program Committee:

Tom Ferrin	Univ. of California, San Francisco
Steve Johnson	AT&T Bell Laboratories
Lou Katz	Univ. of California, Berkeley
Lewis Law	Harvard Univ.
Alan Nemeth	Prime Computer, Inc.
Deborah Scherrer	mt Xinu, Inc.
Waldo Wedel	NBI, Inc.

Tutorial Coordinator:

Michael Tilson            Human Computing Resources, Inc

USENIX Conference Coordinator:

John Donnelly            Denelcor, Inc.

USENIX Meeting Planner:

Judith F. DesHarnais

Proceedings Preparation:

Rob Kolstad              Convex Computer Corp.



# Table of Contents

Thursday, January 24, 1985

## Plenary Session

Thursday (8:30 am – 9:30 am)      Conference Coordinator: John Donnelly, Denelcor  
Program Chair: Charisse Castagnoli, Teknekron Infoswitch  
Program Chair: Rob Kolstad, CONVEX Computer Corporation

Opening Remarks

*Conference organizers and USENIX board*

KEYNOTE ADDRESS: Whither the Gurus

*Rob Kolstad, Convex Computer Corporation*

## Satellite News

Thursday (9:30 am–10:20 am)      Chair: Lou Katz, University of California, Berkeley

Netnews via Satellite: A Progress Report (12/84) ..... 1  
*Lauren Weinstein, Vortex Technology*

Research into Liability Issues in Netnews Transmission .....  
*Susan Nycum, Gaston Snow & Ely Bartlett*

## Kernel Implementation

Thursday (10:30 am – 11:45 am)      Chair: John Quarterman, Univ. of Texas, Austin

Porting the 4.2BSD UNIX Virtual Memory Subsystem ..... 4  
*Jim Mankovich & Rob Kolstad, Convex Computer Corp.*

A Multiple CPU Version of the UNIX Kernel ..... 11  
*Eric J. Finger, Michael M. Krueger, Al Nugent, Masscomp*

Tilde Trees in the UNIX Environment ..... 23  
*Douglas Comer, Ralph E. Droms, Purdue University*

# Languages

Thursday (1:15 pm – 2:50 pm)

Chair: Steve Johnson, AT&T Bell Laboratories

DIBOLIX — An Implementation of DIBOL under UNIX .....	30
<i>Gary Aitken, Kenneth &amp; Christine Scott, Finished Software &amp; SHA Computers Inc.</i>	
Modula-2: An Alternative to C for System Programming .....	34
<i>Morris Djavaheri &amp; Stan Osborne, San Francisco State University</i>	
Concurrent C — An Overview .....	43
<i>N.H. Gehani &amp; W.D. Roome, AT&amp;T Bell Laboratories</i>	
Panel Discussion of C versus Other Languages on UNIX .....	51
<i>Steve Johnson &amp; Michael Powell, AT&amp;T Bell Laboratories &amp; DEC</i>	

# User Interfaces

Thursday (3:10 pm – 4:10 pm)

Chair: Alfred Correia, Computer Thought Corp.

Development of a Compiler for the Bourne Shell .....	52
<i>Vincent Kasten &amp; Paul Ruel, Concentric Associates, Inc.</i>	
Access — A Program to Interpret Pathname Access Permissions for UNIX .....	59
<i>Steven J. Mahler &amp; David A. Curry, Purdue University</i>	
A High-Performance Model for 2-D Alphanumeric Display Generation .....	65
<i>Paul Bame, Hewlett-Packard</i>	

# Performance

Thursday (4:10 pm – 5:15 pm)

Chair: Tom Ferrin, University of California, San Francisco

Monitoring System and Process Performance .....	69
<i>William J. Meyers, SCI Systems</i>	
Interpreting UNIX Benchmarks .....	78
<i>John Saxer, CIE Systems</i>	

Friday, January 25, 1984

## Networking

Friday (8:30 am – 10:10 am)

Chair: Joe Kalash, University of California, Berkeley

Implementing XNS Protocols for 4.2BSD .....	90
<i>James O'Tool, Chris Torek, Mark Weiser, University of Maryland</i>	
UNIX Kernel Networking Support and the LINC Communications Architecture .....	98
<i>Joseph E. Requa, Lawrence Livermore</i>	
Transparent Integration of UNIX and MS-DOS .....	104
<i>C. Kline, G. Popek, J. Rothschild, R. Schulz, J. Uttal, Locus Computing Corp.</i>	
Overview of the Sun Network File System .....	117
<i>Dan Walsh, Bob Lyon, Gary Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss, Sun Microsystems</i>	

## Standards and Directions

Friday (10:30 am – 12:00 pm)

Chair: John Chambers, Microelectronics Center

Latent Source Bugs and UNIX System Portability .....	125
<i>Alan Filipski, Motorola Microsystems</i>	
The Clipboard Data Interchange Facility .....	131
<i>Robert T. Nicholson, Sydis Inc.</i>	
Can't Happen or /* NOTREACHED */ or Real Programs Dump Core .....	136
<i>Ian Darwin &amp; Geoff Collyer, University of Toronto</i>	
A Capability Based Protection Mechanism Under UNIX .....	152
<i>Daniel Klein, The Avatar Corporation</i>	

# Software Tools and Applications

Friday (1:30 pm – 3:10 pm)

Chair: John Trudeau, Teknekron Infoswitch

An Overview of the SETOPT Command Line Option Parser Generator .....	160
<i>Gary Perlman, AT&amp;T Bell Laboratories</i>	
Geritol for Old Programs or Troff's Got a Lot of Life in It Yet! .....	165
<i>Robert Lawson, Avi Naiman, David Slocombe, Mathew Zaleski, SoftQuad Inc.</i>	
Interactive Examination of a C Program with Cscope .....	170
<i>Joseph L. Steffen, AT&amp;T Bell Laboratories</i>	
A Basic Direct Access Method for UNIX .....	176
<i>Robert R. Richards, Chemical Abstracts Service</i>	

## UUCP and Netmail

Friday (3:30 pm – 4:30 pm)

Chair: Tom Watson, Scientific Machines Corp.

Status of the USENIX UUCP Project .....	183
<i>Karen Summers-Horton &amp; Mark Horton, AT&amp;T Bell Laboratories</i>	
A Parser for Electronic Mail Addresses .....	184
<i>Peter Honeyman &amp; Pat E. Parseghian, Princeton University</i>	
Automatic Forwarding of Mail in CSNET .....	191
<i>Michael T. O'Brien, Bolt, Beranek and Newman</i>	
Notesfiles: Why You Should Use Them .....	195
<i>Raymond B. Essick IV, University of Illinois, Champaign-Urbana</i>	

## AUTHOR INDEX

- 30: Aitken, Gary  
65: Bame, Paul  
117: Chang, J. M.  
136: Collyer, Geoff  
23: Comer, Douglas  
59: Curry, David A.  
136: Darwin, Ian  
34: Djavaheri, Morris  
23: Droms, Ralph E.  
195: Essick IV, Raymond B.  
125: Filipski, Alan  
11: Finger, Eric J.  
43: Gehani, N. H.  
117: Goldberg, D.  
184: Honeyman, Peter  
183: Horton, Mark  
51: Johnson, Steve  
52: Kasten, Vincent  
117: Kleiman, S.  
152: Klein, Daniel  
104: Kline, C.  
4: Kolstad, Rob  
11: Krueger, Michael M.  
165: Lawson, Robert  
117: Lyon, Bob  
117: Lyon, T.  
59: Mahler, Steven J.  
4: Mankovich, Jim  
69: Meyers, William J.  
165: Naiman, Avi  
131: Nicholson, Robert T.  
11: Nugent, Al  
90: O'Tool, James  
34: Osborne, Stan  
184: Parseghian, Pat E.  
160: Perlman, Gary  
104: Popek, G.  
51: Powell, Michael  
98: Requa, Joseph E.  
176: Richards, Robert R.  
43: Roome, W. D.  
104: Rothschild, J.  
52: Ruel, Paul  
117: Sager, Gary  
117: Sandberg, R.  
78: Saxer, John  
104: Schulz, R.  
30: Scott, Christine  
30: Scott, Kenneth  
165: Slocombe, David  
170: Steffen, Joseph L.  
183: Summers-Horton, Karen  
90: Torek, Chris  
104: Uttal, J.  
117: Walsh, Dan  
1: Weinstein, Lauren  
90: Weiser, Mark  
117: Weiss, P.  
165: Zaleski, Mathew



# Netnews via Satellite: A Progress Report (12/84)

Lauren Weinstein  
Computer/Telecommunications Consultant  
P.O. BOX 2284  
Culver City, CA 90231  
(213) 645-7200

UUCP: {lhn4, decvax, selsmo, clyde, allegra}!vortex!lauren

About half a year ago, at the Salt Lake City Usenix conference for the Summer of 1984, I proposed a plan for the broadcasting of Usenet netnews materials via television vertical interval data transmission techniques.

Since that time, with the assistance of various organizations and individuals, I've worked to move from that proposal to an experimental/demonstration system embodying the concepts of that plan. Such a system, now dubbed "Project Stargate", has become operational.

I successfully installed UUCP site "stargate" on Monday, December 3rd, 1984. The stargate is a Fortune 32:16 UNIX system, with 30 megabyte disk, which Fortune Systems has graciously made available for the purposes of satellite netnews delivery experimentation. Primary expenses for the experiment are being covered by Usenix, with special thanks due to Lou Katz, who has strongly supported my efforts in this area through thick and thin. Thanks are also in line for Bell Communications Research (BCR), which has provided both moral and tangible support for the experiment. BCR's Brian Redman, Mike Lesk, and Stu Feldman deserve special thanks.

The stargate is located at the primary satellite uplink facilities of Southern Satellite Systems (SSS) in a rural area about 20 miles outside of Atlanta, Georgia. The stargate data transmissions are appearing as a portion of the vertical interval on Turner Broadcasting's "Superstation" WTBS, which is transmitted (uplinked) to satellites via SSS facilities. SSS has very generously made a continuous 1200 bits per second data channel available to us without charge for the experiment.

At the time this paper is being written (December, 1984) WTBS is actually being "simulcast" on two satellites by SSS. The WTBS studio

signal is currently being sent from downtown Atlanta via a dedicated terrestrial microwave link to the rural SSS facilities, then uplinked (transmitted) to one satellite (SATCOM IIIR) from that SSS location. This satellite signal is received by an SSS earth station located back downtown at the WTBS studio facilities, then uplinked via another SSS uplink transmitter (at the same downtown location) to a second (more powerful) satellite (GALAXY I).

This procedure was implemented to allow cable companies time to switch from the older SATCOM satellite to the more powerful GALAXY satellite in an orderly manner, while maintaining the SSS vertical interval data on WTBS on both satellites during the simulcasting period. In early January, 1985, the WTBS signal will be removed from SATCOM and the GALAXY feed will provide all WTBS services. The WTBS signal will be fed directly to GALAXY from the SSS downtown site starting at that time.

Even after this transition, the SSS vertical interval data (including our experimental data) will continue to be available on WTBS (via GALAXY), thanks to another satellite data link between SSS's rural location and the SSS facilities downtown at Turner Broadcasting. Eventually, all of SSS's vertical interval computer equipment, and our own stargate computer(s), will possibly need to be moved to the downtown Atlanta location, but this will probably not need to occur in the immediate future in any event.

WTBS is sent to over 31 million cable TV subscribers across the country via over 8000 cable companies. With the appropriate decoders, the netnews transmissions are theoretically available at almost any point where satellite-delivered WTBS is received, either by cable TV or direct satellite pickup (through inexpensive earth stations) where cable delivery of WTBS is not present.

We are currently in a demo/test mode -- the exact shape of a production system is still under discussion and subject to change. Usenix's support of the current experiment does not in any way obligate them to support a production system, though there are all sorts of favorable possibilities. A production system would require a backup computer and would most likely operate at a data rate ranging from 1200 through 9600 bps depending on hour to hour load factors (that is, we'd probably be running at higher speeds during the evening/night). Since all terrestrial points in range of the satellite transmissions receive WTBS essentially simultaneously, this is a true broadcast medium which makes much greater use of even 1200 bps transmissions. A 1200 bps data stream reaching the entire country at once is a considerable improvement over multitudinous point-to-point phone calls or data links!

I cannot at this time describe a procedure for "signing up" for netnews from Project Stargate -- that will have to wait until, and if, a production system comes to pass. Decoder availability, costs, and various other factors are still being worked out. As a rule of thumb, the basic decoder "package" (commercially manufactured by Zenith) will probably cost approximately \$500, with another \$150 to \$200 or so for a special processing/buffering/interface board to handle message selection, error correction, mainframe buffering/flow control, and similar operations.

The \$500 price for the combination of cable demodulator and vertical interval decoder is about half that of equipment now in use for vertical interval decoding applications. The Zenith equipment includes built-in remote addressing and data decryption facilities. While we have reason to believe that the cost of the Zenith equipment should be in the \$500 area (depending on demand factors) the price listed above for the "special" buffering hardware board is a "guesstimate", since at this time we do not know who would be willing to manufacture such a device and on what terms. We will probably have to design this special hardware largely by ourselves. However, the estimate will hopefully be in the ballpark of the eventual price.

It is also hoped that the possibility may exist for "buy back" and/or rental of decoders and related equipment to further reduce the costs to the end users. A small monthly fee would presumably be assessed for satellite netnews delivery in a production environment, but note that any such fee should be minuscule compared with the phone bills that most sites now pay, or

are likely to soon pay, for large portions of netnews. Please remember, however, that for now this is all simply an experiment and that we are quite some ways from a production system of any kind.

At this time (12/12/84) the stargate is sending status data via satellite/cable TV feed to my own UUCP site ("vortex") on a continuous basis. The observed error rate is quite low at my location. Since the 1200 bps virtual data stream is being decoded from the incoming video at megabit/second data rates, I do not expect to see an increase in overall error rates even if higher virtual data stream rates were in use. Testing has definitely shown the necessity of a buffering/interface board between the decoder and mainframe computers, since even at relatively low data rates, occasional characters are lost when continuous data is fed into many multiuser/multitasking computer systems. However, the need for this buffering hardware in a production system has always been expected.

Test transmission of some netnews materials will commence shortly, via the human "netnews screeners" who have volunteered to participate in the experiment. It is not contemplated that ALL netnews would EVER be sent via satellite, due to technical and broadcast restrictions, but a large (and growing) chunk of netnews is suitable right now, and this is the portion of netnews that can be expected to see the highest growth in the near future. Of course, even assuming a production system does develop, sites will always be free to continue receiving some or all netnews via the conventional dialup telephone network -- no sites will ever be "forced" to join the satellite delivery system.

Submission of materials to the stargate will be via UUCP dialup (using a special encryption technique to avoid "unauthorized" submissions) and control is via a separate dialup line from "vortex" (here in Los Angeles).

The current demo/experimental system, at 1200 bps, is relatively simple since we do not yet have the specialized buffering hardware mentioned above. For now, flow control (such as it is) is being handled through uplink data delays of fixed length. This technique will obviously be used only for the purposes of the experiment and not in any production system.

Outside of testing at my location in Los Angeles, testing will also take place (via a decoder that I'll take/send to various sites) at (most

likely) Lucasfilm, Ltd., NASA-Ames, and various other locations. If all goes well, a physical demo of the system is planned for the Dallas Usenix Conference in January, 1985, via a 9-foot diameter portable satellite earth station to be brought in for the occasion.

By the way, the first message I sent over the Stargate satellite link (sent for about 24 hours, in fact, as a test run) was:

hello, universe

It seemed appropriate enough!

If you're interested in the basic technical details of vertical interval data broadcasting as it relates to this project, please see my paper in the Salt Lake City Usenix proceedings of Summer, 1984. I'd be happy to try answer questions regarding the project where possible, but please don't ask for details regarding a production system, since that's still in the basic planning stages. Also, please make all inquiries via network mail (to my UUCP address) whenever possible, or use the phone as a second choice. Please avoid sending me U.S. mail, since it is by far the most inconvenient with which to deal.

Once again, I'd like to thank everyone who has supported my efforts to make this idea a reality. We've all been donating our time and resources to this project in the hopes that it may pave the way for something truly useful in the near future. Usenix, Fortune, BCR, and SSS have helped us to potentially take a giant step forward toward improved communications, via one of the most widely-distributed basic cable broadcast services in the world. About six months ago, when I first proposed broadcasting netnews via satellite, I never imagined that so much would happen so fast.

--Lauren--

# Porting the 4.2BSD UNIX Virtual Memory Subsystem

James E. Mankovich  
Robert B. Kolstad  
Convex Computer Corporation  
1819 Firman, Suite 151  
Richardson, Texas 75081  
(214)669-3700

## ABSTRACT

UNIX is a general-purpose, multi-user, interactive operating system that is rapidly becoming the industry's standard. Because of these and other characteristics of UNIX, it was chosen as the operating system to be used by CONVEX Computer Corporation and its customers.

This document describes the work entailed in porting the VM subsystem of 4.2 BSD UNIX from the VAX architecture to the CONVEX architecture. It explains the assumptions uncovered about the VAX VM architecture within the 4.2 UNIX kernel as well as how these assumptions were resolved in its port.

## Goals

CONVEX Computer Corporation required an operating system for its new affordable supercomputer. The choice of designing a new operating system was an unpleasant alternative to porting an existing one. Berkeley UNIX, with its virtual memory and other enhancements, became the system of choice for CONVEX. The operating system group committed to an ambitious implementation schedule which required the operating system to be up and running two months after the machine passed diagnostics.

Aside from the overall goal of porting the Berkeley VM subsystem to the CONVEX architecture, several intermediate requirements were established:

- 1 Change as little of 4.2BSD UNIX as possible
- 2 Adhere to the high level paging/swapping algorithms.
- 3 Isolate and remove machine dependencies from the VM Subsystem
- 4 Support 128 Mb of physical memory and up to 2 Gb processes
- 5 Utilize CONVEX hardware 'referenced' bits to enhance paging performance

Possible future Berkeley UNIX releases motivated the decision to change as little of 4.2BSD UNIX since CONVEX may want to upgrade in the future. By modifying only what is required in order to perform the port, it was thought to be easier to incorporate any bug fixes or enhancements made to UNIX in the future.

Since the 4.2BSD UNIX VM Subsystem has been field tested by many sites, we anticipated higher reliability and performance by adhering to its high level paging and swapping algorithms. The isolation and removal of machine dependencies from the VM subsystem was set as a goal because of the anticipated need to port UNIX to yet another virtual memory architecture in the future.

The support for large physical memories and virtual process sizes utilizes the capabilities of the CONVEX C-1 architecture. By utilizing hardware "reference" bits (as opposed to simulating them as is done in 4.2BSD VAX UNIX), it was hoped that system paging performance would be improved.

## CONVEX vs. VAX VM Architecture

Both the VAX and the CONVEX computers have virtual memory architectures with a 32 bit virtual address space. Each implements this space using page tables to perform virtual

address translations. The list below outlines the differences between the physical and virtual memory block size, virtual address translation mechanism, and user/system protection mechanisms:

- o VAX has 0.5 Kb vs. CONVEX 4 Kb pages.
- o VAX virtual memory uses contiguous page tables and segment length registers. CONVEX uses double level page tables with integral "valid" bits.
- o The VAX uses hierarchical processor modes for virtual memory protection; CONVEX uses hierarchical rings.
- o The VAX has hardware modified bits and no referenced bits; CONVEX has both hardware modified and referenced bits.

The total virtual address space on both the VAX and the CONVEX is 4 Gb. The two virtual memory architectures differ in the way in which the 4 Gb address space is partitioned between process and system as well as in the mechanisms used for virtual address translation.

The VAX global virtual address space comprises two 2 Gb address spaces denoted *process* and *system*. The *process* address space further splits into two 1 Gb regions known as the P0 and P1 regions. These three regions (P0, P1, and *system*) define the total extent of software accessible virtual memory on the VAX. Figure 1 illustrates the layout of the VAX global virtual address space.

Figure 1: VAX Virtual Address Space

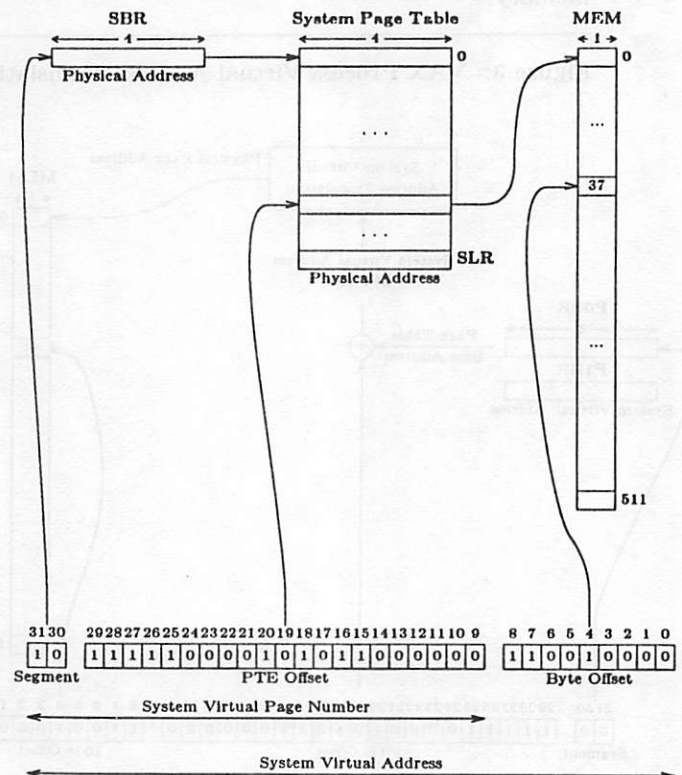
Virtual Address	Virtual Address Space	
00000000	P0 Region Growth Direction ↓	PROCESS
3FFFFFFF		
40000000		
7FFFFFFF		
80000000	Growth Direction ↑ P1 Region	
BFFFFFFF		
C0000000	System Region Growth Direction ↓	SYSTEM
FFFFFFF		
Reserved		
FFFFFFF		

## VAX System Address Translation

The system page table (SPT) which maps the *system* virtual address space is a contiguous array of page table entries (PTEs). The system base register (SBR) contains the physical address of the system page table. The system length register (SLR) contains the number of PTEs within the system page table. Each PTE within the system page table maps 512 bytes of physical memory, so the total extent of system virtual memory is  $512 * c(SLR)$  bytes.

Figure 2 illustrates system virtual address translation. The first 23 bits (2 for segment, 21 for PTE offset) of the virtual address determine the System Virtual Page Number (SVPN). The two-bit segment portion of the SVPN chooses the SBR which contains the physical address of the base of the system page table. The PTE offset portion of the SVPN chooses a PTE from the contiguous array of PTEs stored at the base address. This physical address within the selected PTE forms the all but the bottom 9 bits of the physical address of the data entity desired; the bottom 9 bits are the same ones as the byte offset in the system virtual address.

Fig. 2: VAX System Virtual Address Translation

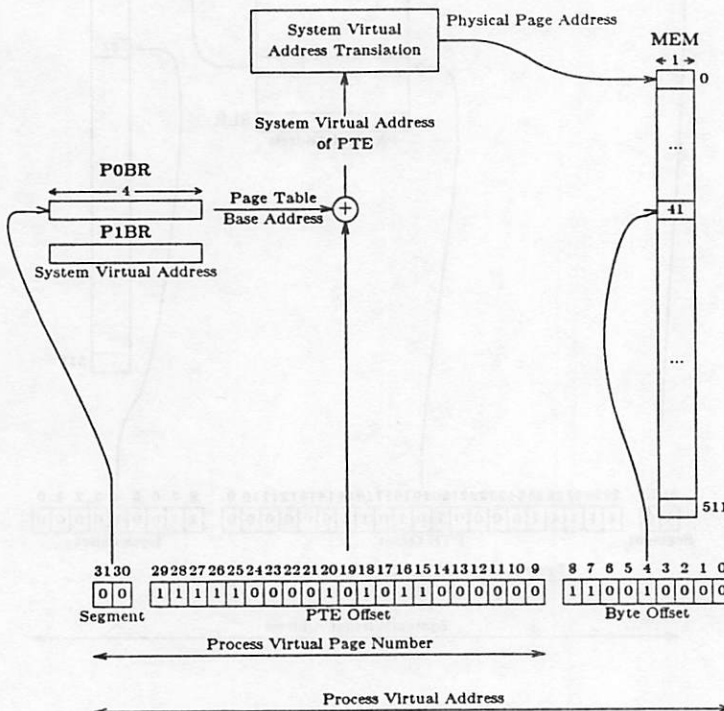


## VAX Process Address Translation

Two independent page tables, P0PT and P1PT map the *process* virtual address spaces called P0 and P1, respectively. The base registers P0BR and P1BR contain the *system* virtual address of the base of the *process* page tables (n.b. the difference between system virtual address, process virtual addresses, and physical addresses throughout this discussion). The length registers POLR and P1LR define the extent of the process page tables. The P0 and P1 regions grow toward each other, the P0 region being used for data and the P1 region for stack.

Since the process page tables exist within the system virtual address space, process virtual address translation is a two level address translation scheme using both system virtual addresses and process virtual addresses. Figure 3 outlines the steps performed in translating a Process Virtual Address. The first step generates the *system* virtual address of the *process* PTE associated with the *process* virtual address. The second step generates the physical address of the addressed entity using the contents of the PTE loaded from *system* virtual memory. This mechanism permits each process page table page (512 bytes) in system virtual memory to map 64 Kb of process virtual memory.

Figure 3: VAX Process Virtual Address Translation



The length registers SLR, POLR, and P1LR control the validity of any virtual address within a region by defining the length of the page tables. Each page table entry specifies the residency and access privileges of the associated virtual page.

## CONVEX Virtual Memory

The CONVEX virtual address space comprises eight 512 Mb segments numbered 0 through 7. Of these 8 segments, 4 define the process address space and 4 define the system address space (though the address interpretation is uniform). Table 4 shows the virtual address space layout of the CONVEX architecture.

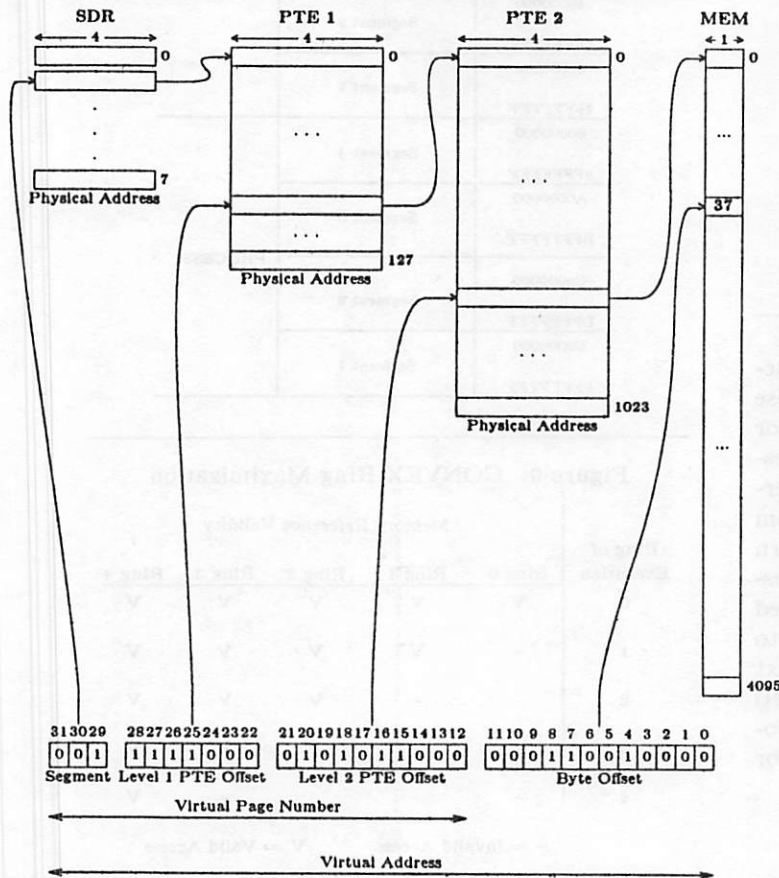
Table 4: CONVEX Virtual Memory Space

Virtual Address	Virtual Address Space	
00000000	Segment 0	SYSTEM
1FFFFFFF		
20000000	Segment 1	
3FFFFFFF		
40000000	Segment 2	
5FFFFFFF		
60000000	Segment 3	
7FFFFFFF		
80000000	Segment 4	PROCESS
9FFFFFFF		
A0000000	Segment 5	
BFFFFFFF		
C0000000	Segment 6	
DFFFFFFF		
E0000000	Segment 7	
FFFFFFFF		

A set of 8 Segment Descriptor Registers (SDRs) map the virtual address space. Each SDR contains the physical address of 128 contiguous PTEs which compose the first level page table for the segment. Each first level PTE points to a contiguous array of 1024 PTEs which is the second level page table. The SDRs control the validity of each 512 Mb Segment; the first level PTEs control the validity of 4 Mb regions within each segment; and the second level PTEs control the validity of each 4 Kb page within the 4 Mb regions.

The second level PTEs associated with the virtual page control the validity, residency, and access privileges of a given virtual address within a valid segment. First level PTEs control the validity and residency of 4 Mb regions; second level PTEs control validity, residency and access privileges of 4Kb pages. Figure 5 depicts the CONVEX virtual address translation mechanism for all virtual addresses.

Figure 5: CONVEX Virtual Address Translation



### Virtual Memory Protection Mechanisms

Virtual memory protection is the function of validating whether a particular type of memory reference is to be permitted to a particular page. The VAX and CONVEX architectures support two types of memory protection:

- o Protection from invalid memory references
- o Memory access privilege verification for valid references

Protection from invalid memory references is a requirement for a robust operating system environment in which the system is protected from process address references. Access protection provide a means for enforcing specific reference characteristics of particular virtual memory pages, e.g. read, write, or execute access.

The VAX architecture uses hierarchically ordered processor modes coupled with PTE protection codes to provide virtual memory protection. The processor can be in one of the following mutually exclusive four modes: kernel, executive, supervisory, or user. kernel mode is the most privileged mode and user the least privileged.

Associated with each virtual page is a PTE protection code which describes the accessibility of the page for any given processor mode. The protections codes permit memory access protection within the following limits:

- o Each mode's access can be read/write, read only, or no access
- o If any mode has read/write access, then more privileged modes have read/write access
- o Read access implies execute access

Figure 6 describes the available PTE protection codes implemented by the VAX architecture.

Figure 6: VAX PTE Protection Codes

MNEMONIC	K	E	S	U
KW	RW	-	-	-
EW	RW	RW	-	-
SW	RW	RW	RW	-
UW	RW	RW	RW	RW
KR	R	-	-	-
ER	R	R	-	-
SR	R	R	R	-
UR	R	R	R	R
ERKW	RW	R	-	-
SRKW	RW	R	R	-
URKW	RW	R	R	R
SREW	RW	RW	R	-
UREW	RW	RW	R	R
URSW	RW	RW	RW	R

- = no access

R = Read Only

RW = Read/Write

K = Kernel

E = Executive

S = Supervisory

U = User

The Berkeley UNIX implementation on the VAX utilizes only 2 of the 4 possible processor modes: kernel and user. This reduces the possible PTE protection codes to the set shown in Figure 7.

Figure 7: VAX PTE Protection Codes used by UNIX

MNEMONIC	K	U
KW	RW	-
UW	RW	RW
KR	R	-
UR	R	R
URKW	RW	R

- = no access                      K = Kernel  
R = Read Only                    U = User  
RW = Read/Write

The VAX has a set of machine instructions for changing "processor modes". These instructions permit changes from one processor mode to an equivalent or more privileged processor mode. UNIX uses the "change mode to kernel" instruction (chmk) to enter the kernel from user space in a controlled fashion. The return from interrupt instruction (rei) causes the processor to revert to an equivalent or less privileged processor mode. UNIX uses this instruction to return to user mode after a system call or context switch. These two instructions (chmk and rei) provide the basis for the UNIX user/kernel protection mechanism as well as the framework for the system call and scheduler implementations.

#### CONVEX Protection Mechanisms

The CONVEX architecture uses a hierarchical ring structure along with a set of addressing rules based on these rings to provide protection between the kernel and user. The eight segments are divided into five rings as shown in figure 8

A set of addressing rules known as ring maximization controls the validity of a memory reference made while the processor is executing within a specific ring. The concept of ring maximization stipulates that the current ring of execution defines the extent of memory access privileges. The hierarchical ordering of the rings dictates that higher priority rings have access to all rings of equivalent or lower priority. Figure 9 shows the validity of memory references given the

current ring of execution.

Figure 8: CONVEX Ring Structure

Virtual Address	Virtual Address Space	
00000000	Segment 0	SYSTEM
1FFFFFFF		
20000000		
3FFFFFFF	Segment 1	
40000000		
5FFFFFFF		
60000000	Segment 2	
7FFFFFFF		
80000000		
9FFFFFFF	Segment 3	
A0000000		
BFFFFFFF		
C0000000	Segment 4	
DFFFFFFF		
E0000000		
FFFFFFFF	Segment 5	
	Segment 6	
	Segment 7	

Figure 9: CONVEX Ring Maximization

Ring of Execution	Memory Reference Validity				
	Ring 0	Ring 1	Ring 2	Ring 3	Ring 4
0	V	V	V	V	V
1	-	V	V	V	V
2	-	-	V	V	V
3	-	-	-	V	V
4	-	-	-	-	V

- = Invalid Access                      V = Valid Access

In relation to the VAX architecture, the rings can be thought of as processor modes with ring 0 being the most privileged mode (kernel) and ring 4 being the least privileged mode (user). The major difference between the CONVEX and VAX architecture is that the PTE protection bits on the CONVEX are not mode dependent. Only memory references which are considered valid by ring maximization proceed to use the PTEs to determine access privileges.

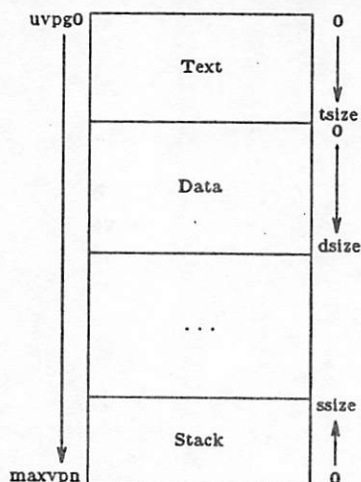
Since the current ring of execution determines access privileges, a ring crossing mechanism exists to protect against illicit ring crossings. A ring crossing can only occur by executing an explicit instruction which crosses rings or by an

exception which requires kernel services. The CONVEX architecture supports a cross-ring procedure call/return mechanism which permits only inward calls (towards ring 0) and outward return (toward ring 4). By using the CONVEX cross-ring call/return mechanism and placing the UNIX kernel in ring 0 and user processes in ring 4, the CONVEX protection mechanism directly maps the VAX protection mechanism.

### Architectural Assumptions within Berkeley 4.2 BSD UNIX

The UNIX VM subsystem partitions a process's address space into three independent segments: text, data, and stack. These segments have respective extents of *tsize*, *dsize*, and *ssize*. See Figure 10 for a visual depiction of these segments. There are two schemes for defining a page's number. One scheme indexes the pages linearly starting with 'uvpg0' and continuing through the final stack page 'maxvpm'. This number is the "process virtual page number". The other scheme numbers the pages within a segment (forwards for text and data, backwards for stack). Each process has a virtual page zero (on the VAX this is 0) where the text starts and a maximum virtual page number (on the VAX this is 0x7FFFFFFF) where the stack starts (the stack grows toward lower memory). Within 4.2 BSD UNIX there is a set of macros (in *vmmac.h*) for translating between virtual page numbers and segment numbers and vice versa.

Figure 10: 4.2 BSD Process Layout



The initial step in porting the 4.2 BSD UNIX VM subsystem was the isolation of the architectural dependencies. One basic premise of the VM subsystem was that a process virtual page

number could be generated given a specific process id and system virtual PTE address. The VAX architecture satisfies this assumption because the process page tables are contiguous within system virtual memory. Subtracting the system virtual PTE address from the base of the process page table in system virtual memory generates the zero relative process virtual page number. Virtually all the VM routines within VAX UNIX pass a PTE address and count as parameters. If a subroutine needs a process virtual page number, the PTE address and process id easily generate it.

The CONVEX double level page table address translation scheme does not easily permit virtual page number generation from PTE addresses because the second level page tables can reside anywhere within physical memory. Without forcing all process page table pages to be contiguous on the CONVEX architecture, it is very difficult to generate the virtual page associated with the PTE from given a PTE address and process id. Since non-contiguous process page tables were desired (i.e., page table pages were to be allocated the same way as general memory pages), CONVEX UNIX could have no assumptions about contiguous page tables.

The next step (and most difficult part) of the VM port was the removal of the assumption that a virtual page number could be generated from a PTE address. Removing this assumption required two changes. The first was the removal of a set of macros in *vmmac.h* for converting a PTE to a virtual page number. The second was to change the VM subroutines which used these macros to accept the virtual page number as an argument. By passing only virtual page number between the VM subroutines, those routines which required PTE addresses could easily generate them. After removing the PTE to virtual page translation assumption, it became much easier to port the VM system to other architectures since there is always a way to generate a PTE address given a virtual address.

The use of VAX modified bits constituted another machine dependency. On the VAX architecture, the PTE which points to a physical page contains that page's modified bits. VAX UNIX deals with the modified bits on a virtual page basis since they are associated with a PTE on the VAX. The CONVEX modified bits are not contained within the PTE's but rather are located within the memory mapped I/O portion of memory. In order to deal with the modified bits independently of virtual pages, CONVEX UNIX uses five new macros to perform all modified bit

manipulation. The five macros are:

- o `modified` - Returns the state of the modified bit
- o `setmodified` - Set the state of the physical page to modified
- o `clearmodified` - Set the state of the physical page as unmodified
- o `anyclmodified` - Is any page within a cluster modified
- o `distclmodified` - Distribute modified bits to all pages in a cluster

These macros remove the architectural dependencies of VAX UNIX. Their correct definition allows the VM code to be used on any VM system.

### Implementation

Because hardware for the CONVEX computer was not to be available for several months, CONVEX augmented the instruction set simulator constructed for validating compilers. Adding a complete simulation of the CONVEX virtual memory system required less than two working days. Simulating the operating system required about 20 minutes of CPU time to reach the first prompt (from the shell spawned by `init`). The first system message contained an interesting twist on the nUxi problem: "Unix Version 1" printed as "xinUreV nois 1" due to the 32 bit word transpositions between the simulator and the simulated I/O system.

The virtual memory project required about five man-months to complete. Most routines were unchanged; only two required major modification. The system is now in production use and appears to be reliable after several months of testing and internal use.

### Vector Register Scheduling

Since the CONVEX architecture provides integrated vector processing, the vector registers (8 registers, each 128 elements by 64 bits) have to be shared among all the processes. In order to reduce the overhead incurred on saving the vector registers in memory when context switching, the C-1 hardware provides a system trap for vector register access.

The vector register access trap is controlled by two instructions, one to enable or disable a vector valid trap, and one to test the current state of the vector valid trap. When vector traps are enabled, any access to the vector

registers cause the kernel to be entered. When vector traps are disabled, access to the vector registers proceeds normally.

The CONVEX UNIX kernel uses the vector valid trap mechanism to defer the saving and restoring of the vector registers until absolutely necessary. Also if a process does not use of the vector registers, no vector registers saving is done.

When a process is scheduled for the first time, it is started with vector valid traps enabled. Only when a process takes a vector valid trap is the process considered a vector program, and vector register scheduling required. If a vector valid trap occurs for process A and process B is currently holding valid vector state in the vector registers, the vector registers are flushed to process B's u area and then allocated to process A which trapped. Process A is then scheduled immediately with vector traps disabled so it may use the vector register.

### Conclusion

The CONVEX UNIX kernel uses a new style of virtual memory addressing with double indirect page tables which allows the second level page tables and the data pages themselves to be paged out. A modified system with generalized virtual memory macros is arguably more "portable". This kernel repairs several virtual memory latent bugs and has been run for several months in a beta-test environment.

# A Multi-CPU Version of the Unix Kernel

## Technical Aspects and Market Need

Eric J. Flinger  
Michael M. Krueger  
Alan F. Nugent

MASSCOMP Engineering  
Westford, MA

### ABSTRACT

This paper describes an implementation of a Unix kernel that supports multiple CPU's. There is only a single copy of the Unix kernel and each of the processors path it independently. The kernel is written entirely in "C" (and a small amount of 68K assembler) and is based on a previous version of MASSCOMP Real-Time Unix (RTU) which is a derivative of SYSV rel2 with Enhancements from BSD4.N. A more powerful version of an MC500 system is created by simply adding in another CPU and some memory (and optionally an additional hardware floating point processor), effectively doubling the compute capability, without any OS alterations. It encompasses changes to deserialize the kernel, hardware and software implications of going MP, and associative modifications to drivers, utilities, and applications. Another feature of our approach is that no agent processes or multiple copies of drivers are used at all. Performance characteristics are also discussed as well as the impact of MP systems on various application arenas. Some of the topics include: Hardware/Software overview, Design Objectives and Constraints, Background information (prior attempts i.e. Purdue hack), Actual Design Techniques, Implementation Details, Support Consideration and Comparative Performance Statistics.

### Introduction

#### Hardware Overview -- Backplane and Busses

MC-500 systems all share the same backplane arrangement insofar as the system and memory are concerned. A standard Multibus is used for peripherals such as discs, tapes, ethernet, and the like, while a MASSCOMP extended form factor card is used for graphics and data acquisition processors. The P2 connector is used for a MASSCOMP proprietary high speed (8MB/sec) Memory Interconnect Bus (MI) that couples CPU, 1 MB ecc memory cards, Hardware Floating Point Processor, and Array Processor together. There is one contiguous 15 Multibus arrangement while there are two MI's in each 15 Slot MASSCOMP system. This creates a hardware asymmetry that requires the software to fully coordinate cache's and Address Translation Buffers resident on discrete CPU in each MI.

Having 2 Memory busses does provide a good deal more positive impacts than negatives however. For instance, having 2 MI's decreases

the traffic you would expect to find on a single symmetric backplane and creates an inexpensive multiprocessor because you don't have to build a very high bandwidth common bus. Although the CPU's reside in separate busses they can see all of memory. The penalty for accesses across the MI's is not significant, so a process running on one CPU can be accessing memory remotely (local to the other CPU).

The Multibus has also been "super-setted" by MASSCOMP to provide two times the normal bandwidth of the bus for MASSCOMP peripherals. Called a block mode transfer, the system sends two 16 bit quantities across the bus at the expense of one clock cycle (100ns), effectively doubling the bus throughput into the 4-6MB/sec range.

#### Hardware Overview -- CPU

The CPU module manufactured by MASSCOMP is of the extended Multibus form factor. The board contains a Motorola 68010 and a Motorola 68000 both clocking at 10 Mhz. The 68010 (Executor) runs all user and system code

and is the "main" processor, performing all tasks excepting Translation Buffer maintenance and FP emulation. The 68000 (Fixor) wakes up when the 68010 faults and does an instruction backoff. The only code the 68000 runs fixes the Hardware address translation buffer (ATB) during a fault and hands the process back to the 68010. If the system has a Hardware Floating Point Processor, the Fixor also performs floating point emulation in the absence of the hardware to eliminate the need for recompiling applications code. The CPU Board also has a 4KB cache, a 1024 x 4KB entry Address Translation Buffer to map virtual addresses to physical pages, and a 1024 x 4KB entry Hardware I/O map to allow DMA into a virtual environment.

A multi-processor MC-500DP uses 2 of these CPU modules, one in each of the memory interconnect busses. The only differences between the modules is the absence of the clock driver chip, a pal change that programs the "slave" to ignore all interrupts except those a level 6, and a new arbiter PAL that solves bus arbitration problems that could occur when processors reference each other's memories. The CPU modules are also jumpered to respond either as a "master" or a "slave".

## Hardware Overview -- memory & other

MASSCOMP memory boards contain 1MB of ECC memory and are of the extended form factor. The system currently supports 8 MB of memory in a MC-500DP and 6 MB in a MC-500. The Virtual Address space of the machine is 16MB. MASSCOMP also manufactures a Hardware Floating Point Processor that sits in the MI and runs about 900K whetstones/sec, a Hardware Array Processor (MI device) that performs a 10 Megaflops, a Data Acquisition and Control Processor which extends a third bus out of the machine (STD+) and runs at 8 mips, and a variety of Graphics Processors, in color and monochromatic, all with 68000's with resolutions from 640x480 to 1024x800.

Packaging options for the MC-500DP include a vertical 15 slot "Workstation" cabinet or a horizontal rack-mount or tabletop 15 slot configuration. At the moment the kernel supports all MASSCOMP peripherals including 2 Hardware Floating Point Processors (one per MI) and the Hardware Array Processor (one only).

## Software Overview

MASSCOMP RTU (Real-Time UNIX) was originally a derivative of Bell SYSTEM III. The OS was then enhanced with BSD style virtual memory management and copious utilities from BSD as well. For example, a full job control C-Shell is supported. Since the mainstream of MASSCOMP's business is in the scientific and engineering communities our UNIX was modified to perform like a real-time operating system. Those modifications included:

- Changes to the scheduler to allow for absolute, unchanging priorities,
- Modification of the file system to allow for contiguous disc files.
- Provisions for locking down (make them unswappable) all or part of a process.
- Addition of AST (Asynchronous System Traps), the software analog of a priority hardware interrupt.
- Alterations to device drivers to allow transfers to the "raw" device (without going through kernel buffer space)

With the 2.2 Release of RTU the kernel is now a derivative of SYSTEM V Release 2 complete with semaphores, message queues and dynamic shared memory segments. The OS has also been embellished with BSD4.2 style Local Area Networking (sockets and the like). A great number of utilities from both UNIX camps are still provided. The 2.2 Release of MASSCOMP RTU also brought support for multiple CPU's in a single MC-500 chassis.

While we are on the subject of modifications/enhancements, the standard compiler fare from Bell/BSD were, at best, unsuitable for most heavy scientific users. MASSCOMP has dedicated a great deal of internal resources in the way of providing high performance compiler technology without compromising UNIX "standards" or flexibility. An example of this is shown by the increase in whetstone floating point performance that was achieved through compiler changes alone.

## Opportunities

Under normal conditions for a user to increase the available compute power at an installation, he/she must purchase another complete system. The incremental costs for such a move are considerable; such costs discourage users who cannot afford to double their system expenses and

system administration problems. Such a user may well be satisfied with the capabilities of a given system solution (in our case an MC-500) yet conclude that the purchase of a higher-capacity single processor product is the only cost effective alternative.

There is also a category of potential MASSCOMP users who have a need for dedicated processing by a M68010-based processor, yet who do not wish to completely forgo the benefits of the UNIX environment. A solution for these customers can open up new sales opportunities, especially for OEM applications.

### Response

It is now possible to create a more powerful version of the MC-500 system that will supply as much as 95% more performance yet costs less than 20% more than the original system. This paper describes this the software implementation of this product, the MC-500DP, which satisfies through enhancements to the UNIX kernel, the requirements for a more powerful member of the MASSCOMP product line.

For the timesharing environment, the MC-500DP provides a broader processor resource base for user process execution. Although performance results will be sensitive to system call loading and I/O traffic characteristics, most timesharing applications will see significant performance improvements.

The dedicated processor requirement will be well served by the MC-500DP also. The second processor may be dedicated for certain processing activities yet still have access to UNIX services when needed.

The challenge of the development was to create this dramatically improved environment with existing MC-500 hardware and a reasonable upgrade to RTU software.

Simply put, the outcome was to not only create a system that would exhibit a transparent access to a multiple cpu resource, but also to provide control and a more sophisticated user interaction as well.

### Objectives

## Compute-bound Tasks

Higher system throughput for tasks that are generally known to be compute-bound, that is to say, tasks that are nearly 100% user code. If the task is floating point intensive, so much the better. The MC-500DP can be configured with 2 Hardware Floating Point Processors for twice the performance in a floating compute intensive tasks. Some of the performance figures later shown put this application throughput at 1.9 to 2.0 times that of a single processor MC-500.

## Timesharing Performance

Higher system throughput for process mixes that comprise a balanced load (system vs. user processing). This type of load can be found in many timesharing systems. It will be shown that commonly encountered workloads may show an increased throughput of up to 1.5 to 1.7 times that of an equivalent MC-500.

## Custom Application Performance

Higher system throughput for applications that have been purposefully divided into components that take advantage of a dual processor architecture. Examples of this had been proposed by several OEM customers and end user prospects. It is possible to get performance factors greater than 1.7 for well-designed applications of this type.

## Ease of Use

Easier to administer compared to two independent systems providing equivalent throughput since no duplication of the file system occurs.

## Costs

Significantly less expensive, both in purchase price and maintenance charges (also compared to two systems).

## Reliability

A lower failure rate due to the use of fewer total hardware modules (compared to two systems).

## User Control

A process should have some control over its execution context (i.e., on which processor(s) it will be able to execute).

## Administrative Control

The operating system will provide some general controls on the use of the processors, independent of the preferences specified by processes.

## Constraints

### Performance Minimum

An MC-500 running a RTU release that contains MC-500DP support (release 2.2 or later) does not suffer a significant performance loss due to the existence of dual processor support in the operating (It is clear that there will be some minor lengthening of some common code paths in the kernel, and this will affect the speed of those paths. This is an unavoidable result of the increased capabilities of RTU. Other sources of kernel performance degradation are (1) the need to turn off cacheing of kernel data pages, and (2) the overhead associated with multiple TB invalidation logic.)

The development for this product was begun as more or less a test bed for the feasibility of provide a Multiple CPU environment in the next generation MASSCOMP Computers. The performance goals for the project were never tremendously optimistic. Much to our delight the system kept performing better and better and the impact of de-serializing the kernel was not felt by other parts of the executive. The performance figures that appear later are a great source of pride for the system developers.

Easier to administer compared to two independent systems providing throughput since no duplication of the life system

### Transparency

For the casual user, operation in the MC-500DP environment must be transparent (compared to operation on an MC-500).

### Functional Compatibility

All normal functions of RTU (release 2.2) must be preserved.

### Integration

A lower failure rate due to the use of Very active development of RTU release 2.2 was underway throughout the duration of this project. There were also bug fixes to the kernel and utilities which could have had catastrophic effects on the DP support, not to mention, performance. It is an understatement to say that the two parallel efforts integrated with good results. This project also had to be a logical intermediate

step in the development of symmetric multiprocessor software for the next generation architecture.

## Background

At the time we began development there was not any standard approach to multiprocessor operating system software. Without this precedent we at once had little existing code to build on, yet had much flexibility in determining what it was we should do. Several major development factors are changing as we move into dual-processor and multiprocessor software.

### Lack of Available Software

UNIX System code written for multiprocessor systems is not commercially available. This was a big change from the way RTU kernel development had previously worked, since much of what we have today is an amalgam of pieces available from various sources (Berkeley, AT&T), and we expected to see the going be a little tougher and slower. A more realistic fear was the greater risk of conceptual mistakes, not having created a concurrent UNIX kernel.

### Lack of Active Development Community

For the timesharing environment, the base for user process execution. Although performance was good, we knew that moving in multiprocessor UNIX, we would step out on our own into an operating system area that has few precedents and fewer successes that point the way. Since the competitors who are pursuing their own flavors of multiprocessor UNIX are playing it close to the vest, it was difficult to assess the technical merit and competitive position of our work. We therefore needed to use an aggressive operating system approach that was a serious attempt to become a leader in multiprocessor operating system technology.

### Lack of Clear Future Directions

The process of designing a multiprocessor UNIX is fraught with the risks of establishing a direction that will not find favor with the evolving UNIX system community, regardless of whether that mainstream is "the right thing". There are many issues that arise for the first time, and there are relatively few in the UNIX systems industry who are actively attacking these problems (though §4.2 identifies competitors in multiprocessor systems). As multiprocessor systems (and high-reliability systems) become more common, significant diversions from the traditional UNIX architecture will occur. We need to balance

innovation with awareness of ongoing work in the multiprocessor UNIX area, being careful of whose bandwagon we might end up on.

## Conclusion

It became clear that we must engineer a superior multiprocessor UNIX system, while basing it on largely untried software technology. We knew that we must therefore build in safety nets and prototyping that would allow us step-wise refinement of the product. It was crucial that we base our final product on *coherent* actions rather than *random* reactions. This is why the MC-500DP is a very valuable step on the way to the complex multiprocessor operating system technology we intend to have in the next 12 to 18 months.

## Related Work

### External Projects

The dual-processor VAX-11/780 work at Purdue University by G. Goble has proven to be an interesting study, yet it is not especially applicable to the problems faced in implementing the MC-500DP. We have the disadvantage of asymmetric hardware, and the advantage of more options in the operating system software.

In the Purdue system, there is a very strict division between master and slave in that *all* kernel functions are performed on the master. The slave processor runs a non-trivial amount of customized assembler code that services its own clock, watches for interactions with the master, and finds work to do in the queue; this code is not common to corresponding code executed by the master.

The Purdue system has clear design and implementation directions that reflect the assumptions that no more than two processors will be used, and that they will be VAX-11/780 processors.

### Internal Projects

Some work had been done at a design level that addressed the ultimate needs of concurrency in the kernel for graphics and array co-processors. The sleep/wakeup work was especially well-suited with the dual-processor code at an early stages.

## Competitive Data

There is not much known about the work being done by other companies interested in multiprocessor UNIX variants. The following is an overview of the official and unofficial competitive information we have (Information on competing systems is earnestly solicited.).

### VAX 11/782 Multiprocessing System

This system is a dual-processor VAX system that is presented in some detail in a recent publication from Digital Press. Because it is so well documented, it and the Purdue work for which we have sources are the two systems that can be compared to our work in the greatest detail. The hardware characteristics of interest are the following:

- The processors are tightly coupled by sharing one to four MA780 shared memory controllers. This device has the ability to implement inter-processor interrupts, something sorely lacking in the Purdue hardware.
- One processor is the master, and owns all peripherals. The MA780 tracks the cacheing being done by each connected processor, and selectively sends cache invalidation messages to the proper processor.

Several items in the software design are very worthy of note:

- A design goal in the VMS modifications was to not affect the size or speed of VMS on uniprocessor systems. In the long run, they went through fantastic gyrations to prevent the addition of a mere 8KB to the kernel. The extra code for multiprocessor operation is loaded into non-paged pool, is linked in to the kernel on the fly, and duplicates existing routines in the kernel. Probably the hardest thing to understand about why they did this all is the removal of the 8KB and the links to it if you shut down the slave (even temporarily).
- The slave does absolutely *no* processing other than user mode execution. They do not distribute the kernel services or driver functions to the slave. This will tend to give that system the kind of balance problems the Purdue system encounters.
- The master does all of the process selection for the slave. This is different from

both Purdue (in which the slave runs a special *findslaveproc()* routine to get a process) and different than our approach (in which the slave can execute the scheduling portions of the kernel).

Overall, the VAX-11/782 does not appear to be a step toward a symmetrical multiprocessor operating system. That fact may or may not mean anything in terms of future product offerings that could be expected.

## LOCUS

This is a distributed architecture based on a loosely coupled network. It has its roots in public-domain software out of University of Southern California.

There is some element of process migration for the purpose of distributing the system load. From the kinds of discussions at SIGOPS last fall, it is not clear when LOCUS will be available commercially. The initial target machine was the VAX, and there are rumors that work is underway to rehost the system to 68000-based systems. LOCUS was presented at the June USENIX conference in Salt Lake.

## HP-9000

This product from Hewlett-Packard runs a sub-kernel on as many as four processors, and achieves a degree of parallelism in the UNIX environment that is layered upon the sub-kernel. The HP-9000 is a symmetric multiprocessor system.

## AT&T

There is now a member of the 3B20 family of computer systems that is a dual-processor configuration. The primary use of this architecture is performance. Latest information from those at USENIX is that they have implemented a multiprocessor System V kernel based on semaphores, and claim a 1.6 to 1.9 performance factor. A big question is whether sources for such an implementation will soon find their way into AT&T release tapes for UNIX. Note that their system does not have paging, graphics, or real-time capabilities.

## Options

The available courses of action centered on some variant of the master/slave concept used in the Purdue system. There were two extremes

in the design spectrum.

## Strict Purdue Imitation

The first idea was to stick closely to the Purdue architecture, building the customized assembler code and cleanly dividing the master and slave at the system call boundary. The problem with this approach is that the differences between the MC-500DP hardware and that of the Purdue VAX are quite significant (memory symmetry is the biggest issue), so the Purdue model is a long way from being where RTU must be in terms of resource management and scheduling policy.

## Full Kernel Concurrency

At the other end of the spectrum, more than a little consideration had been given to installing as much locking and concurrency in the kernel as possible. The idea is to build something that is as close to the eventual multiprocessor next generation software as possible.

## Design

The MC-500DP is a tightly coupled asymmetric dual-processor system. In this system we capitalize on the existence of multiprocessor support in the CPU module, then implement some creative (but not massive) changes to RTU that will support the use of two processors.

## Hardware Asymmetry

The access to memory by the two processors is not uniform because of the delays associated with crossing the break between the two MIs. There is also the fact that the two processors have their own translation buffers and memory caches. From a hardware point of view, these two sets of caches are fully independent, so the operating system must carefully coordinate their use.

## Software

The RTU operating system software remains the basis of the system. Fundamental changes to the kernel occur in several areas.

## Master and Slave Paths

A large portion of the kernel runs concurrently, primarily in *clock()* code and *swtch()* code. The former has to do with periodic scheduling activities, and the latter is software that manages the state of processes in the system. The

net effect is to allow the slave to acquire user processes for execution.

### System Service Processing

The slave selectively hands off designated system call processing to the master. Some system services, however, are performed on the slave, in particular the very simple ones that do not block and do not touch many data structures that would require locking. The benefit of this approach is that there will be less context switching to the master.

### Policy for Asymmetry

As noted in the hardware asymmetry section above, memory access in the MC-500DP is not symmetric, since it takes longer for a CPU to access memory that is not located on its MI. There are, therefore, significant implications for the memory resource allocation policy in RTU, since a process will run faster if its pages are on the same MI as the processor that is executing its code.

### Multiple TB Invalidation

Invalidation of the translation buffers on the two processors is coordinated in software. There is no hardware linkage that performs this function. The TB cache is invalidated by the associated fixor processor.

### Multiple Cache Invalidation

The memory caches on the two processors must be set properly for each memory page, in that both processors may not cache any page that contains read/write data.

### Visibility and Control

When the project began, RTU had no features that provided control and visibility for a dual-processor system; these were designed and built. Some existing utilities were expanded, and there was a need for some new utilities.

### Software Implementation

#### Boot ROM

At time of hardware reset, the boot ROM code operates out of the cache memory until it turns on memory. It initializes the fixor and leaves the TB disabled. The next things to happen related to MC-500DP support occur in *start()*.

### Low Level Kernel Paths

The master begins executing the *start()* code, and after some very basic initialization, pokes each possible slave CPU ID to see who is there. If a slave is present, it initializes (including priming its clock) and responds to the master through a shared memory location then enters *switch()*. There it blocks on the global run/sleep queue lock.

Meanwhile, the master sees that the slave is there, and records the fact in global kernel data. When all slave processors have been poked, the master continues on to do the rest of kernel initialization (allocating data, initializing drivers, etc.). Then the master clears the run/sleep queue lock and the slave falls through the rest of *switch()* to *idle()*, where it executes a stop instruction. At this point, the slave will be awakened by either an interrupt from the master, or a DUART interrupt from one of the on-board serial lines or clock. Either event will cause the slave to pass through *switch()* looking for work out of the run queues.

Note that beginning with the *start()* code, the slave and master are executing the same kernel. This is quite distinct from the Purdue approach wherein the slave had its own specialized copies of low-level routines to process clock interrupts, acquire user processes, and hand off system service processing. This design direction in the MC-500DP kernel is a very important step toward the eventual symmetric multiprocessor operating system that will be needed on the next generation.

### Clock Interrupt Processing

The slave and master each have their own clock. The MC-500DP does not need the "accelerated clock" hack that the Purdue system uses (The Purdue system does not have any inter-processor interrupt capability, so the master uses non-cached global variables to send requests to the slave. A faster clock rate is used (8 times normal tick rate) so that the slave processor can more quickly poll these control request flags.) since the master can communicate with the slave through the IPIR interrupt. Note that the master and slave clocks are of course not synchronized, so *clock()* processing occurs at different times on the two processors.

The *clock()* routine is completely executed by the master, but only parts are executed by the slave. The master and slave both compile statistics for the current process, update CPU usage

data, and perform profiling for the user process (if profiling is in progress). All of these activities are done on both processors because each processor contains the necessary state information which is invisible to another CPU.

## Processor Switching and Trap Handling

Unlike the Purdue system (in which *all* system calls are handled by the master), the slave hands off system calls to the master only when necessary. Some system calls are executable on the master only, but some are executable on the slave as well. The allowable processors for a system call are indicated in a new entry in the *sysent[]* table which is the "*dispatch mask*" for the service.

In an even greater departure from the Purdue architecture, the execution of a system service can flop back and forth at will between the slave and the master. This creates many opportunities to spread the system load across the two processors.

Some services that are more simple (such as read-only access to kernel data with no *copy-out()* activity) will remain on the slave for processing. The 70 or so RTU system services fall into a spectrum of level of difficulty; about 40 of the services have been converted for execution on the slave.

Therefore, some of the more difficult system services are partially executable on the slave, which further softens the distinction between master and slave. This option is attractive because it reduces the interruption of the master for system services in cases where the system call can complete on the slave. Frequent interruption of the master processor is a major problem in the Purdue system, to the point that scheduling policy in that kernel penalizes processes that do "too many" system service requests.

Note that traps and page faults are always referred to the master for processing via a trap dispatch mask that contains only the master processor's bit.

The mechanism for processor switching goes as follows: Each process has an *effective processor select mask* which is a field of four bits, one per CPU. These bits indicate the set of processors on which the process is willing to run. The *switch()* code ANDs the mask with the executing processor's corresponding bit (remember that *switch()* is executed by all processors). If the result

is *true*, the process can be executed on the processor, and the processor "picks up" the process and proceeds to run it. Note that this may be user processing or system processing in the context of a user process.

So a process can be handed off to another processor by simply setting the appropriate bits in the processor select mask and entering *switch()* via *qswitch()*. For example, if a system call initiated by a process on the slave is tagged as "master only", the processing is moved to the master by saving the process's effective CPU select mask, altering it to be "master" only, then calling *qswitch()*. The process next wakes up on the master, where processing continues, perhaps to the end of the system call. When system call processing is complete, the saved effective CPU select mask is restored, *qswitch()* is called, and the process is next executed on one of the processors allowed by its select mask.

## Cache Control

Since the memory caches are not coordinated in hardware, steps are taken in software to prevent problems with stale cache data. Each processor can cache:

- Kernel text.
- Pages that are marked "globally cacheable" at the time they are assigned to the a process. These may be either text or data pages for any process running on either processor.

Note that this more flexible caching policy regains much of the performance penalties imposed by the asymmetry of the hardware. This caching capability is the single greatest performance factor in the MC-500DP operating system.

## I/O Drivers on the Slave

A major move towards greater symmetry in the kernel was to move some of the I/O driver activity over to the slave. There were numerous considerations that involved: directing hardware interrupts to the slave instead of the master, locking of appropriate kernel data structures, and effects on latency in the slave, to name a few.

## User Devices on the Slave MI

User-provided devices may be placed on the slave MI, but the conditions under which they are accessed is restricted.

A device may be mapped with the *pmapi()* system service. The process that is doing the mapping may be running on either processor, although remote memory references will be much longer.

Interrupts from such a device must occur at interrupt level 6, the same level as the FP and AP on the slave MI (if they exist).

### Data Acquisition and Control Processor Support in MP

Since the DA/CP is a Multibus module, we can tie it to the master in the normal way. Note, that because the slave is an inherently low-latency environment and is very clean for preemption, processes running there will see superior AST delivery times.

There are some other ideas that can enhance DA/CP and real-time performance in the MC-500DP. An *mpadvise()* option has been created to allow a superuser process to quiesce the clock on the slave, which eliminates all asynchronous disruptions of the slave processor that don't involve requests of the master (which happens at the initiative of the user process anyway).

### Utilities

At least two existing utilities, *ps* and *w*, have been upgraded to show new information related to the processing contexts of processes they examine. The shells have new commands that control the processor select masks.

The *vmstat* utility has been converted to display the additional kernel statistics on the slave processor operation.

A system administration utility has been created that effectively allows access to many of the functions available in *mpadvise()*.

## Multiprocessor Resource Management

### Introduction

This section presents the problems of resource management in multiprocessor systems, describes the policy of this product, provides details the implementation, and concludes with a description of the user controls available in the MC-500DP.

## Resources

The system must allocate and control the following resources:

- Processors; in this system there are two processors, master and slave; the clock on the slave processor is a separately controllable resource
- Memory pages; they exist on both MIs, and are closely associated with either the master or slave processors.

Processes are assigned to processor resources for execution, in both user and system contexts.

Process virtual pages are assigned to physical memory page resources.

### Policy Options

At first blush, it might seem that processes and virtual pages could be randomly assigned on the basis of what resources are most free. This can not be done due to asymmetries in the system. One reason that random assignment is insufficient policy is the existence of an application in which the user needs to override the tendency of such a policy to spread the processing load. In some applications, a program must run with as little interruption as possible on a processor that is dedicated to one purpose.

The problem of providing controls over these resources is divided into several parts:

- At the lowest level, there are software mechanisms that provide the control.
- At an intermediate level, there are the panel switch and system configuration options that establish the initial conditions of those mechanisms.
- At the highest level, there are on-line facilities to adjust resource allocation policy on the fly.

### Objectives

The following are the broad objectives in the resource management area:

- Provision of general-purpose modes. There will be four modes of system operation will be designed to cover most of the normal uses of the system.
- Flexibility for special applications. The resource management behavior will be

highly modifiable for the benefit of certain applications.

- Proper carry-through of defaults. Once defaults are established, either by selection of a general-purpose mode or through on-line adjustment, the management of resources will conveniently propagate in the system.
- Control at system generation. Initial resource management behavior can be set up in the configuration file, so that the resulting system has the desired behavior at boot time.
- Control at boot time. Initial resource management behavior can be at least partially determined at boot time by both panel switch positions and options in the RTU boot command line.
- Control on-line. Resource management behavior can be controlled during system operation.

## Modes of Operation

There are three primary modes of operation in which the system initially runs. The MODE panel switch, the mode parameter in the configuration file, and the RTU boot command line all act together to determine which mode the system starts up in.

**MASTER.** This setting is to be used by those who wish to run all processes on the master processor.

**SLAVE.** This setting is to be used by those who wish to run all processes on the slave processor.

**DUAL.** This setting is to be used by those who wish to run all processes on either processor at the operating systems discretion.

## Effective and Default Values

The internal mechanisms for processor and memory resource management rely on the concept of *effective* and *default* values. An effective value is being actively used in the decision being made. A default value is not actively used, but is used to initialize the effective value. This concept primarily applies to the mask values belonging to each process, but also loosely applies to the mask values in the kernel as well.

## Online Control of Resource Management

The *run* command is a interface program that allows the user to create a process that has different CPU and policy select values than the parent shell process.

The *mpadvise()* system service is a mechanism by which most of the resource management mechanisms may be adjusted while the system is running.

The *run* command is implemented with the *mpadvise()* call. The *run* program processes the switches and constructs the corresponding *mpadvise()* call to set the correct policy select and cpu select values. Then the *exec()* of the remainder of the command line is done, and that process inherits the masks.

## Performance

To this point we have alluded to performance ratios in this paper. We must point out that this system promotes it's efficiency as the result of parallel execution. While it does not notably increase the efficiency on any single-process task, it may greatly improve the efficiency of any task that take advantage of separate processors. The MC-500DP boosts the aggregate computing power available from a single MC-500 chassis.

The slave processor is best suited for compute intensive tasks of response-time-critical tasks that can profit from a low overhead environment. In a Timesharing environment the OS handles the distribution of the workload between master and slave rather nicely (in DUAL mode).

A major gain for time-critical processes in MASTER and SLAVE modes is that processes such as init, getty, cron, and update, as well as the daemons and the single user shell, automatically run on one processor, so that the other processor can be reserved for specially selected tasks.

To measure the preformance throughput of the system we ran several "benchmark" type programs, but illustrated here are three of the most universally known and accepted benchmark programs: a spice run, whetstones, and linpack.

SPICE is a large FORTRAN application that makes intense use of floating point operations for general-purpose circuit simulations. System performance figures based on SPICE (or its derivatives) are occasionally seen in advertising and industry performance comparisons. Attached

is a chart that presents the floating point throughput of five computer systems, as measured by SPICE running under UNIX operating systems.

LINPACK is a floating-point-intensive benchmark program written in FORTRAN, and is based on the LINPACK Subroutine Package. Attached is a chart that presents the floating point throughput of 21 computer systems, as measured by the LINPACK program.

There are two ways to construct the LINPACK program. The "vanilla" approach is to compile the standard FORTRAN source set for LINPACK. Benchmarks run with the resulting executable image appear on the right side of the attached chart, labelled "FORTRAN BLAS". (BLAS stands for Basic Linear Algebra Subroutines.) The second way to construct the program is to code these frequently-called subroutines in assembler, optimizing for speed. The resulting runs appear on the left side of the attached chart, labelled "Coded BLAS".

The Whetstone benchmark is very intensive in floating point operations, and is usually written in FORTRAN. Each system's performance is measured in thousands of Whetstones per second (KWHETS/SEC). There are single and double precision versions of the benchmark: the attached chart presents data for single precision.

The MASSCOMP numbers were produced on a MC-500DP system built from standard hardware, running stock RTU release 2.2 with an alpha release of the 3.1 compilers. The results for other systems are drawn from a variety of public sources.

The object of this system comparison is throughput performance, a method frequently used in multiprocessor product evaluations.

The MC-500DP Dual Processor system is one of the first commercially available multiprocessor UNIX-based systems. The MC-500DP system used in this test has two processor modules and two floating point accelerator modules. Throughput results on this system were obtained by simultaneous execution of two of the above named programs.

With the new optimizing FORTRAN 77 and C compilers and the advent of Dual Processor, MASSCOMP systems demonstrate throughput higher than every VAX regardless of

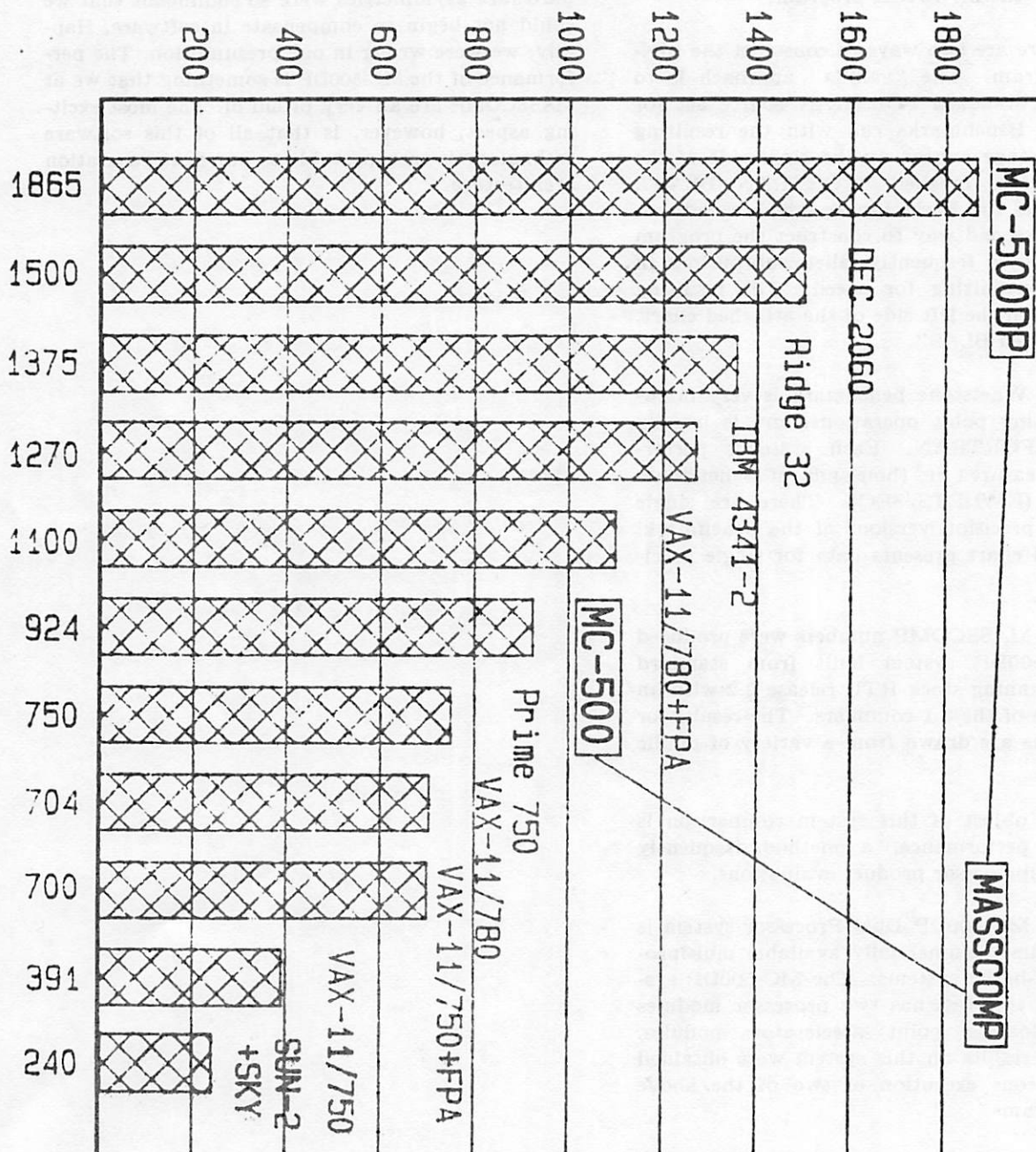
floating point accelerator (FPA), operating system (VMS or UNIX), and model (up to and including the 780).

## Conclusions

In drawing to a close we would like to point out that the performance of any system, MP or not, is really what counts. All during the development of this kernel, we believed that the hardware asymmetries were so significant that we could not begin to compensate in software. Happily, we were wrong in our presumption. The performance of the MC-500DP is something that we at MASSCOMP are all very proud of. The most exciting aspect, however, is that all of this software technology is transportable to our next generation architecture.

# WHESTONES PERFORMANCE

## Floating Point Throughput (KWHETS/SEC)



## Tilde Trees in the UNIX<sup>†</sup> Environment

*Douglas Comer*

*Ralph E. Droms*

### ABSTRACT

TILDE is a multi-year research project exploring distributed computing in an environment where the primary computing engine consists of a cluster of UNIX-like systems loosely coupled with high-speed local area networks. The goal of TILDE is to extend UNIX to provide a transparent integration of local and remote facilities, allowing the user access to remote files and services without forcing the user to know about the location of data, processes, or other objects such as servers. One important aspect of TILDE is the Tilde naming scheme. The TILDE approach to naming objects in a distributed system provides a transparent, consistent mechanism for referencing both local and remote objects.

The Tilde naming scheme substitutes a collection of directory hierarchies, known as *Tilde trees*, for the single UNIX directory hierarchy. A program executes in an environment composed of a *forest* of Tilde trees, which can be selected from the collection of Tilde trees known to the distributed computing system. In the prototype, each of the Tilde trees is mapped onto a UNIX subdirectory. As in UNIX, files are named either relative to the current working directory, or by a full pathname. Full path names begin with tilde, and the first component identifies the appropriate Tilde tree, while the remainder of the pathname completely specifies the file within the Tilde tree. (The naming scheme can be viewed as an extension to C-shell naming mechanism.) A particular process executing within the TILDE system need not know the details of the storage organization or file location in the distributed system; rather, there is a uniform mechanism for searching and accessing Tilde trees.

This paper describes the naming scheme and a prototype implementation. It discusses alternative designs and compromises imposed by the limitations of a single-processor UNIX environment, as well as the impact of using the Tilde environment for tasks such as software development. We show how our Tilde naming software makes it possible to move large programs from one directory to another or from one UNIX system to another without recompiling (even if path names have been bound into the code). Finally, future directions for this work,

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.

including extension to collections of UNIX systems and modifications to include the Tilde tree naming system in the UNIX kernel, are also discussed.

## 1. Introduction

TILDE is a multi-year project exploring distributed computing in an environment where the primary computing engine consists of a cluster of UNIX-like systems loosely coupled with high-speed local area networks. The project addresses many research issues, including naming objects in a distributed environment. The goal of TILDE is to extend UNIX to provide a transparent integration of local and remote facilities, allowing the user access to remote files and services without forcing the user to be aware of the location of data, processes or other objects such as servers. Naming is the key issue involved in making access transparent. In order to provide flexible, convenient and portable names, the notion of a single, rooted directory tree must be abandoned. For example, solutions that impose a virtual root above all UNIX machines (e.g., [1, 2]) are nontransparent because they imply that users understand locations of files and must change software when data or programs referenced by that software moves. We propose instead a new scheme that permits large, complex software systems to be constructed in such a way that the directory in which the system resides can be moved without requiring programs to be recompiled.

The conceptual change from UNIX to our new naming scheme is simple. UNIX employs a single directory tree per machine, with authority for names in the upper part of the tree vested with the system programmer. Our new scheme employs a set of independent trees, with authority for the structure of each tree vested in its owner. We claim that:

*By replacing the UNIX tree-per-machine naming system with multiple independent trees, we will integrate local and remote names into a single mechanism, improve software portability, and retain the advantages of a hierarchical directory system.*

Our new system substitutes a collection of directory hierarchies, known as *Tilde trees*, for the single UNIX directory hierarchy. A program executes in an environment composed of a *forest* of Tilde trees. Each user maintains a *private forest*, which can be thought of as an active subset of all Tilde trees available throughout the distributed computing system. Each Tilde tree has a name, selected by its owner when the tree is created. Thus, a private forest is represented by a set of tree names. The user's private forest establishes the Tilde environment for any processes executed by the user.

Running programs, including command interpreters (which are called *shells* in UNIX), reference files using names of the form:

`~tree/path`

where *tree* corresponds to the Tilde tree in which the file is located, and *path* gives a complete path from the root of the tree to the file. For example, the name "`~dec/bin/xinu`" references a file named *xinu* in directory *bin* located just under the root in Tilde tree *dec*. A particular process executing within the TILDE system need

not know the details of the storage organization or file location in the distributed system; rather, there is a uniform mechanism for searching and accessing Tilde trees. Tilde trees may, in fact, be distributed across a network transparently to the process. The process knows to name a particular file by its Tilde name, regardless of whether the Tilde tree containing that file exists on a local or a remote system.

## 2. The Tilde Tree Abstraction

A distributed computing environment such as TILDE provides access to objects on many computing machines. The naming policies and name interpretation mechanisms in a distributed environment limit the extent to which programs, processes, connections, and data can be exchanged meaningfully. Moreover, poor names may impact program portability or may restrict the usefulness of a system merely by making it inconvenient to exchange objects or share computing services.

In distributed environments, the question of naming is central to system design. A good naming scheme permits efficient exchange of objects by permitting processes to exchange object names. However, names can only be exchanged freely among parts of the system that dereference them the same way. We capture this measure of exchangeability by saying that naming is *transparent* if all names are known globally (i.e., if a given name refers exactly to the same object independent of the context in which it is interpreted). Name transparency is desirable, at least for names that users learn and manipulate, because it decouples object management from object naming.

Transparency can be achieved by making all names unique. Uniqueness requires either a central authority to designate all names, or an agreement by which individual "sites" (e.g., a computing mechanism) can assign names. The establishment of a central naming authority was appropriate when computing systems were independent. However, a central naming authority becomes inefficient and a bottleneck in a distributed system. A central naming authority also reduces the flexibility and imposes a rigidity of structure on distributed systems. All modifications to the system structure and naming conventions must be registered with the central authority. Finally, naming provided by a central authority will not be convenient for users and will not meet the users' needs for short, mnemonic names.

Distributing the authority for name assignment introduces conflict possibilities because multiple sites may choose the same name for different objects. One way to solve the conflicts is to use location dependencies. For example, each site can be assigned a unique site identifier, denoted *siteid*, and assign names as a pair (*siteid*, *otherid*). In practice, the resulting name is only valid as long as the referenced object remains at its original site; moving the object requires inventing a new name.

Site identifiers have been used to extend UNIX file names to a multiple-machine environment in [1, 2]. Both systems overload the syntax of UNIX path names, relaxing the usual interpretation. IBIS prepends machine names onto file names, making names take the form *machine:file*. Names in which the first component does not end with a colon are assumed to be local file names. The Newcastle connection [1] also provides an extension to UNIX by allowing names of the form *"/..siteid/path"*. Names that do not begin with *"/.."* are interpreted with respect to the local system, so all existing software continues to operate correctly. Because the reference *"/.."* does not occur in standard use, there is little possibility for inadvertent reference to a remote file. Note, however, that both examples employ non-uniform names for local and remote references.

In addition to syntactic inconsistencies, location dependent schemes tie the naming of objects to the physical organization of the distributed system, much the same as early file systems tied the name of a file to the physical storage device on which it resided. A connection between the hardware organization and the naming scheme is inflexible because it means that references to named objects must be changed whenever an object moves from one machine to another.

The Tilde naming scheme solves the problem of naming in a distributed environment in a novel way: it replaces a per-machine name binding mechanism with a per-user name binding mechanism. Thus, names are relative to the individual, not to the machine on which the computation is performed. In a TILDE system, users can refer to a file without knowing the file's physical location or the location of the computational service they invoke. Furthermore, a default set of Tilde name bindings, established when the user logs on the system, allows users to refer to system directories and to each other's files almost exactly as they would on a single UNIX system. More importantly, the Tilde naming scheme makes local and distant file names uniform and transparent, allowing consistent access to files on both the local and distant computing mechanisms.

### 3. The Prototype System

We have implemented a prototype of Tilde naming system that executes on a conventional UNIX operating system. The prototype consists of a layer of software that runs outside the kernel, but captures and interprets system calls (e.g., `chdir`, `creat` and `open`). The goal of the prototype was to create an execution environment capable of supporting the typical edit-compile-test program development cycle. As an experiment, we converted a subset of the UNIX commands to run under our Tilde environment.

The prototype, which currently executes under 4.2bsd UNIX, can coexist with UNIX. It maps Tilde trees onto UNIX directories and even allows the user to specify that UNIX path names are allowed. Our prototype includes the following components:

- *A modified C shell* that supports the user interface to the Tilde environment.
- *A modified C library* that contains, in addition to conventional C library routines, a set of procedures that correspond to UNIX system calls.
- *Modified utilities* that have been converted to use the Tilde naming system.
- *A forest of Tilde trees* that provide a UNIX-like for Tilde system program execution

The components of the prototype combine to form an experimental environment in which users can:

- *Initiate a session* as though logging into a TILDE system
- *Create software* that uses the Tilde naming scheme and takes advantage of the resulting portability and access transparency
- *Change their private Tilde forest*

The next sections describe the prototype in more detail.

**Modified C Library.** The new C library includes procedures that intercept system calls, map references to Tilde names onto UNIX names according to the user's Tilde mapping, and call appropriate operating system routines to access files. The new

library is constructed so that programs can be prepared to execute in the Tilde environment merely by linking them with the new library.

The prototype passes the user's Tilde tree name bindings to subprocesses through the UNIX environment. *Tilde tables* are files of mappings from Tilde names onto UNIX names. These files contain pairs of bindings. They are represented as ASCII text according to the simple syntax:

```
~tilde-name | /UNIX-path-name
```

where the symbol "|" is used to separate the Tilde tree name from the UNIX directory to which it is bound. For example, the standard system Tilde table TILDE\_SYS includes the entries:

```
~bin | /usr/tilde/bin
~tmp | /usr/tilde/tmp
~etc | /etc
```

Because they contain ASCII text representations, Tilde tables can be manipulated just as any other text files using utilities such as *vi*, *sed* and *grep*. The environment variable TILDE\_PATH defines a list of Tilde tables which are to be used in looking up a Tilde name. When a Tilde name is encountered, the Tilde environment software performs a linear search of the Tilde tables in the order specified by TILDE\_PATH to find the UNIX name bound to the Tilde name. For the sake of efficiency, an in-core Tilde directory is constructed from the environment variables at the time of the first Tilde reference. Since the in-core directory is constructed at the time of the first reference to a Tilde name, a process can give itself a trusted Tilde environment by making a call to a new procedure *resetpath()* that resets the Tilde directory and TILDE\_PATH to a set of trusted system Tilde tables.

*Modified C shell.* We modified the C shell to:

- process names that begin with "~" according to the Tilde naming system,
- initiate a Tilde system session without logging out of, and UNIX,
- include builtin commands to manipulate the Tilde environment.

The Tilde csh presents users with an environment closely analogous to the initial UNIX system environment when they begin. For example, ~userid refers to the home directory of the user with login id *userid*, just as with the conventional csh.

The Tilde csh allows simulated log in making it possible to switch to a Tilde environment and experiment without permanently changing login shells. The Tilde csh initializes its TILDE\_PATH to include the system Tilde tables TILDE\_SYS and TILDE\_USR. As mentioned above, TILDE\_SYS includes standard system mappings. TILDE\_USR includes mappings of *userid* to home directory for all system users, providing a mechanism similar to the UNIX csh tilde shorthand naming. To simulate login, the Tilde csh changes its working directory to the users home directory, and maintains an internal copy of the name of its current working directory. This current Tilde directory is passed to any subprocesses through the TILDE\_CWD environment variable. Once logged in, a user invokes commands as usual, except that all full paths must begin with "~tree". Tilde csh expands arguments into full Tilde names as well, so "echo ~dec/bin\*" produces names like "~dec/bin/myspell" and "~dec/bin/manual". The Tilde csh allows modification of its environment variables to allow the user to dynamically alter the tilde mappings. To keep the in-core Tilde

directory up to date, it is reinitialized after every modification of an environment variable with the prefix `TILDE_`.

*Modified commands.* We converted a subset of the utilities provided by the UNIX system to use the Tilde naming system. Many of the conversions involved only changing all UNIX system names to Tilde names, and recompiling using the Tilde library.

*Default Forest.* The Tilde forest is composed of directories dedicated to the Tilde prototype and directories shared with the standard UNIX system. For example, `~bin` maps to a new directory that holds the modified commands. Similarly, `~lib` and `~tmp` map to new directories which we created just for the Tilde prototype. Some directories must be shared, however, with the existing UNIX system. For example, `~dev`, `~usradm` and `~etc` are bound to UNIX directories `/dev`, `/usr/adm` and `/etc`, respectively. Thus, Tilde prototype utilities such as `df` and `who` continue to function normally concurrently with the UNIX system.

*Current prototype status.* The Tilde execution environment has been used to bootstrap a copy of all the prototype software. Temporarily, full UNIX path names are allowed to ease the migration to the Tilde prototype (they can be disallowed by the user). The prototype reports warnings of UNIX path name references. The user can suppress warnings setting environment variable `TILDE_WARN` to "NO". Setting environment variable `TILDE_ALLOW` to "NO" disallows any full path references. Tilde trees, as implemented in the prototype, do not fully conform to the abstract definition of Tilde trees. The major difference is that the prototype's Tilde trees are not entirely disjoint, due primarily to the mapping onto the UNIX file hierarchy. The prototype maintains its notion of current working directory based on the Tilde name specified by `cd`, and does prohibit exiting the root of a Tilde tree by changing to directory `..`. No effort is made, however, to determine if a Tilde tree is a subset of another Tilde tree, or if subtrees of different Tilde trees are connected by links.

Another restriction on users of the Tilde prototype is the small subset of the UNIX utilities which has been converted to date. For example, we have not yet tackled the conversion of EMACS to the Tilde environment. We expect to convert more utilities after expanding the Tilde prototype to include network file access.

#### 4. Experiences with the Tilde System

Most of the work done under the Tilde prototype has been software development. We are currently using the Tilde prototype to work on the next revision of the system itself. During development of the Tilde prototype's user environment, we discovered many instances of the close connection between the UNIX file system naming scheme and the UNIX utilities. We attempted to construct the Tilde forest to mimic the UNIX file hierarchy as closely as possible by defining Tilde trees such as `~bin`, `~lib`, `~tmp`, etc., that correspond to existing directories in the UNIX file hierarchy. We expected that this correspondence would allow easy substitution of tilde names for UNIX names in the utility programs (e.g., globally substitute `~bin` for `/bin`), and would ease the conversion of existing software. Unfortunately, UNIX software derives many names in non-obvious (and generally undocumented) ways. For example, the loader `ld` searches libraries in directories `/usr/lib` and `/lib`. This search is performed by first constructing a string `"/usr/lib/libname.a"` and attempting to open the file. If the open fails, the pointer is simply incremented by 4 (to point at `"/lib/libname.a"`), and the open is attempted again. Merely replacing `"/usr/lib"` with

"usrlib" does not produce the intended result. Csh makes use of the shared string generation package xstr. To prevent certain strings (which are dynamically altered during csh execution) from being shared, they are declared as arrays of characters:

```
char pstr[] = { '/', 'b', 'i', 'n', ... 0 };
```

(xstr does not recognize such a declaration as a character string). Of course, a global replacement of "/bin" by "bin" also fails to replace these declarations correctly. Much effort was spent in tracking down these declarations and converting them to tilde names.

## 5. Future directions

One project of immediate interest is the incorporation of the Tilde naming scheme into the make utility. We expect to take advantage of the Tilde naming scheme concept to allow the compilation of one set of source code (maintained in a single Tilde tree) into executable code in several Tilde trees (e.g., one Tilde tree of code for each machine in a network). We also realize that there is work to be done to make the interface to the user's Tilde environment more convenient. Utilities for easy modification of the Tilde environment are under development. The long range goal of this project is to incorporate the Tilde naming system into the UNIX kernels of the machines in the TILDE computing engine. The next step is to move the translation step of the Tilde naming scheme into the kernel. We expect, as a transition step, to build a system which supports both a standard UNIX directory and a collection of Tilde trees, perhaps as entries in the root directory. Finally, we will incorporate a remote file access system to provide user access to remote Tilde trees in a transparent fashion.

## 6. References

- [1]Brownridge, D. R., L. F. Marshall and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!", *Software Practice and Experience*, Vol. 12 (1982), 1147-1162.
- [2]Tichy, Walter F. and Zuwang Ruan, "Towards a Distributed File System," *Proceedings of the Summer USENIX Conference* (June, 1984), 87-97.

# DIBOLIX - An Implementation of DIBOL under UNIX

Gary Altken  
Finished Software  
PO Box 72  
Collbran, CO 81624

Christine Scott  
Dept. Math & Computer Sc.  
Skidmore College  
Saratoga Springs, NY 12866

Kenneth Scott  
SHA Computers Inc.  
RD #3, Box 51  
Saratoga Springs, NY 12866

## ABSTRACT

DIBOL (Digital's Business Oriented Language) was introduced in the early 70's to provide a suitable language for on-line business applications on small (16K memory) mini-computers. The current number of systems using DIBOL is in excess of 50,000, however these are almost exclusively DEC systems running under RT-11 or RSTS/E. The authors wished to run an existing large DIBOL application under UNIX and decided that the best way to achieve this was to develop a DIBOL language capability rather than to rewrite the application. DIBOLIX, the resultant product, is a DIBOL to C translator plus a DIBOL run-time library of C subroutines. DIBOLIX accepts DIBOL code with no re-coding requirement. The resulting C code can be linked with other C subroutines thereby strengthening the capabilities of the programmer. The problem of file name specifications is handled through a run-time look-up table. DIBOLIX has superior performance characteristics to DIBOL.

## INTRODUCTION

It is currently fashionable to talk of UNIX as the operating system of the future for business applications. The flexibility, universality, and variety of programming tools available in the UNIX environment promise much. In addition the relatively low cost of software in this environment provides a strong incentive for businesses to consider the adoption of a UNIX based system.

The authors are among those who believe that UNIX provides much opportunity for business, and based on this belief we decided some time ago to adopt UNIX as our next operating system. We particularly were attracted by the availability of low cost database, word processing and graphics packages on relatively low cost M68000 based hardware configurations. Yet at the same time, as providers of business software for the past decade, we perceived that a move to a UNIX based system could be difficult, and perhaps costly.

Our applications packages had been developed over a period of years using the DIBOL language (Digital's Business Oriented Language). This language was developed at Digital Equipment Corporation (DEC) by Cohen [1,2]. The current specification is a hundred programs and

over a million lines of source code, represented a substantial labor investment in coding as well as a significant investment by our users in implementing the code. This code was running under the DEC RT-11 and RSTS/E operating systems on PDP 11/34, 11/70 and LSI-11 machines. Any move to UNIX could only be done if such code could be ported without rewriting. To achieve this we decided to create a DIBOL language capability for UNIX.

As users of DIBOL we are part of an estimated 50,000 plus installed base worldwide. Like most of these users we use DIBOL as a programming language because it provides an easy to use, high level business programming capability. This is clearly a business language that needs to be accommodated by the UNIX community, if these existing installations are going to opt for UNIX on their next system upgrade. We believe our implementation of DIBOL under UNIX, which we call DIBOLIX provides an effective bridge for such users.

## DESIGN OBJECTIVES

To meet our various requirements for a DIBOL language capability we adopted the following design objectives for DIBOLIX:

1. It should be easily portable from one UNIX system to another. At the time of starting the project we had not decided on which version of UNIX we would use nor which hardware. We also wished to have flexibility in this choice in the future. This clearly dictated that DIBOLIX should be written in C.
2. It should compile and run existing DIBOL source code without rewriting such code. This dictated that any system dependent features would need to be handled automatically by the conversion process.
3. It should give run-time reporting features consistent with that of the DEC implementations. In particular error reporting should be at least as good and should appear the same to the user.
4. It should provide access to the UNIX environment through subroutine call access to C coded routines. Such access to routines written in other languages has not always been available to DIBOL users. (Those who have integrated non-DIBOL routines into their DIBOL packages would need to rewrite this code in C)
5. It should provide a continuing DIBOL development capability under UNIX. While the authors do most of their applications development in UNIX using C, a certain amount of maintenance work is required on existing DIBOL code. Other users might wish to continue to program in DIBOL.
6. Performance of the resulting DIBOLIX package should be good. This is an ambiguous statement in that performance depends greatly on hardware and the version of UNIX one uses. Our interpretation of this requirement is discussed below.

## DESIGN OVERVIEW

We implemented our design objectives by creating two distinct sets of C code, a translator and a run-time library. Each program is translated first into C and then compiled and linked with the run-time library to provide an executable program. Using this strategy we are able to minimize the amount of programming necessary to provide the implementation while at the same time achieving the following advantages:

- integration of C routines into the package is easy
- improvements in the C compiler give improvements to the DIBOL programs

At the same time, this approach gives a slower generation of the executable code than might be possible with a compiler.

The translator is a program which takes as input a DIBOL language source file and generates two output files. The first of these is a C language program which is equivalent to the procedure division of the DIBOL program. The second of these is an assembly language program which corresponds to the data division of the DIBOL program.

Since our intent is to provide a DIBOL language capability, the C code generated by the translator is intended only as an intermediate step in producing the executable code. This intermediate code is not normally seen as it is erased in the sequence of commands that generates the executable code. However it is available and potentially could be used to optimize particular routines in DIBOL if desired.

Initialization of the data space used by the program is done through a program written in the assembly language of the target machine. This was done because the DIBOL data spaces are generally quite large and limitations on the C compiler would prevent use of a C coded routine. To maintain portability of the code this code is generated by a C language routine which references a machine language template for the specific machine.

The run-time library is a library of C routines which provide the features of the DIBOL language. This library consists of routines to handle all the arithmetic and logic functions of DIBOL as well as to service the various subroutine calls which are provided with the language (the XCALLs). It is analogous to the DIBOL run-time system in a DEC implementation but differs in one important respect: Because UNIX does not provide shared libraries, these routines become part of the executable program. This means that programs are larger and consequently are slower in loading.

The separation of the code into these two components has the advantage that improvements in the run-time system can be implemented by merely re-loading the compiled DIBOL code with it.

A significant problem in moving from a RSTS/E or RT-11 system to UNIX is the difference in the file structures. A DEC file name is of the form DEV:[proj,prog]FILNAM.EXT.

Such names are embedded in DIBOL source programs. To allow the translation to handle such names (as well as terminal and printer designations) a table was created, which is referenced during execution, which maps such names into standard UNIX references. Using different tables in different directories allows considerable flexibility in system configuration.

The DIBOL language implementation supplied by DEC provides, through XCALLs (subroutine calls to external subroutines included in the run-time system), a number of system dependent features. These features have been implemented in DIBOLIX so as to allow translation of existing code without modification of the source code. In instances where the XCALL is meaningless in the UNIX environment a null routine is provided.

DIBOL includes capabilities for accepting one character at a time from the terminal (the ACCEPT command). To implement these under UNIX it was necessary to go into cbreak mode, which then implied that a small terminal interface module needed to be created to handle various control characters.

A significant feature of DIBOL (as with many other business languages), is that numbers are stored as strings of characters rather than in a binary representation. An important reason for this is that the data structures allow for multiple definition of elements of the data space, including simultaneous definition of a field as numeric and alphanumeric. Additionally, DIBOL allows a number to have up to 18 digits which is too large for binary coding in a 32 bit long integer. The implication of these statements is that whenever arithmetic or logical operations are performed on numbers they must be performed byte by byte on the character string, or alternatively the string must be converted to a binary form, manipulated and then converted back to a string. This causes such processes to be slow, as indicated below in the performance section. Because such processes are significant in their impact on performance we continually seek to improve our algorithms. Of particular interest in the M68020 processor is the possibility of using the binary coded decimal instructions together with the pack and unpack features of the chip.

## PERFORMANCE

DIBOLIX has been implemented on the NCR TOWER running under NCR's version 1.02 UNIX release which is a system III

implementation. The performance results reported here relate to that implementation.

Our concern for performance--that it should be good--translated in application to meaning that we should be able to move from a PDP 11/34 running RSTS/E version 7.0 to the NCR TOWER with at least no degradation in performance and preferably some improvement. The PDP 11/34 had 128KW of memory and a 300MB Ampex 9300AQ drive with an Emulex SC21 controller (an RM05 equivalent subsystem). We refer to this configuration below as the "PDP". The TOWER has a Motorola 68000 chip, 1 MB of memory and drives an AMPEX 9300AQ drive with a Xylogics 450 controller. We refer to this configuration below as the "TOWER". Our performance tests therefore matched the CPU and operating systems against each other--the drives were the same.

Program size is of concern: Large programs use disk space, more memory during execution and take longer to load. The use of a 32 bit processor versus a 16 bit processor has an immediate effect on the size of programs. More significantly, the lack of shared libraries creates very large executable files even for small programs. On the other hand, the ability to have non-overlaid executable programs (which are bigger in memory) is considered to be a big advantage because swapping of overlays is not needed. A program which does nothing is 5 blocks on the PDP and 33 on the TOWER. A large program that takes 221 blocks on the PDP takes 497 blocks on the TOWER. Conversion of some binary numbers which are currently implemented as integers, to shorts, should reduce sizes significantly.

We examined operational performance in two ways: We were concerned about disk performance and also about execution of DIBOL arithmetic and logic instructions.

Disk performance was tested in a number of ways using a random read test programs and then using a program which did a file compress by deleting every other record in a 2,000 record file. The time results were as follows:

Test	PDP	TOWER
random read	25 millisec(*)	10 millisec
file compress	97 seconds	57 seconds

(\*) this is the best achieved. The result could be twice as large depending on the location of the file

on the disk. UNIX was not sensitive in the same way) These excellent test results are reflected in system performance with applications programs.

A number of logic and arithmetic operations were tested by looping them through several thousand operations. The following results were obtained:

Test	PDP	TOWER
increment	900 microsec	700 microsec
greater than	1200 microsec	1100 microsec
subtract	2300 microsec	3000 microsec(*)

((\*) This procedure currently has some subroutine calls which add to the execution time. We anticipate that by the time this paper is delivered the subtract time will be less on the TOWER than on the PDP) We expect that the performance of the system for arithmetic and logic operations will be better on the TOWER than on the PDP when final tuning is completed in early 1985.

Our experience in installed sites using actual application programs is consistent with these results. Programs generally run faster on the TOWER than they did on the PDP. If the program is disk intensive this improvement in performance is significant.

## CONCLUSIONS

DIBOLIX is an effective implementation of DIBOL in a UNIX environment. Its existence provides a suitable bridging capability for those wishing to take existing DIBOL applications onto a UNIX system. Additionally, DIBOLIX provides a capability for business users to move to UNIX without developing a C programming capability. Current performance of the code meets the initial design objectives of the project.

Future development of the package will include continued improvements in the performance of the run-time library, reduction in the size of disk files, and implementation under UNIX System V. Extensions of the language to take advantage of certain C language features are also being contemplated.

## BIBLIOGRAPHY

- [1] Cohen, J.B. COMMERCIAL APPLICATIONS--WHICH SYSTEM?, IEEE Northeast Electronics Research and Engineering Meeting, NEREM Record, Tech Appl Papers, v 12, 1970
- [2] Cohen, J.B. DIBOL-COMMERCIAL APPLICATIONS FOR THE PDP-8, Papers and Presentations of Digital Equipment User's Society, Spring Symposium, 1970.
- [3] DIBOL-83 LANGUAGE REFERENCE MANUAL, Document Number AD-U0660A-TK, Digital Equipment Corporation, Maynard, Massachusetts.

# **Modula-2**

## **An Alternative to C for System Programming**

*Morris Djavaheri and Stan Osborne*

Computer Science Department  
San Francisco State University  
1600 Holloway Avenue  
San Francisco, California 94132

### **ABSTRACT**

The Modula-2 programming language is the most recent language to be developed by Niklaus Wirth. A number of implementations now exist for Unix systems. The Modula-2 language is described by contrasting it with PASCAL and C. An implementation of Modula-2 for the MC68000/Unix environment is discussed. Examples of Unix/Modula-2 programs are given. The goal of the paper is to show that Modula-2 can be used as the alternative of choice to the C programming language for Unix system programming projects.

### **1. Modula-2 History and Overview.**

The Modula-2 programming language [Wir80] is the most recent language development of Niklaus Wirth who designed it in large part to answer the special needs of system programming. As a direct descendant of PASCAL [Wir71] and MODULA [Wir77], it inherits basic language features of PASCAL and most of the design philosophy of MODULA, which it also superseded. In five primary areas Modula-2 is an advancement over PASCAL.

- **Module Structures** — All components of an application are contained in programming structures called modules. A simple program may consist of only one module. On the other hand, using modules to divide a large, complex programming problem into its logical components makes the process of designing and implementing large programs easier to manage. (The name of the language is derived from this feature.)
- **Separate Compilation** — Each Module may be compiled separately, allowing flexibility in project management and program design. Also, Modula-2 has the ability to specify all uses of global information in separate definition modules. The "Definition Module" feature can be used to hide the implementation of a module from a programmer thereby providing an extra level of protection from system bugs. It allows access to the functionality of the module without knowledge of how the functionality of the module is implemented. Code and data are shared or hidden by selective

use of the "IMPORT" and "EXPORT" declarations. IMPORT allows a module access to code and data global in the system. EXPORT is used to declare something as global to the module definition.

- **Increased Transportability** — Machine-dependent modules should comprise a small part of a system's overall code. Modula-2 implementations support procedures written in a machine's native language. By isolating all machine dependencies to a few modules, only those modules need to be modified when a program is transported to another Modula-2 implementation.
- **Multi-Processing Primitives** — The ability to create multiple processes and to transfer execution to them as a co-routine is coupled with the ability to implement an interrupt service routine. Since processor priorities may be specified as part of a module's implementation, these features allow a pre-emptive scheduler for multi-processing to be easily implemented.
- **Concise, Flexible, and relatively O/S independent** — Many awkward PASCAL language features have been cleaned up in corresponding Modula-2 implementations. Procedures now can operate on "open" lengthed arrays, related declarations may be grouped for better program clarity, compilers are allowed to optimize the evaluation of conditionals, O/S independent I/O libraries now are implemented in Modula-2, and fewer keywords are required for the use of many language statements. Modula-2 begins with a very small support requirement (about 1K bytes) which allows it to be used in the smallest of system programming applications.

### 1.1. Differences between Modula-2 and C.

Because these two languages do not share a common design history they have significant differences. The more significant of these differences are described here:

- **Strong type checking** — Both C and Modula-2 provide for strong type checking. With C the type checking operation is optional and a separate program must be run to perform the checking (lint). With Modula-2 type checking is done every time a program is compiled. If explicit type-conversion is needed in a Modula-2 program, it is done with deliberate use of conversion functions. In C, this is done with the explicit type-conversion operator called "cast".
- **Scope and Side-effects** — In C every procedure is global to the whole program. There is no protection from side effects. Also, a C procedure may not have a subroutine for its own private use. In Modula-2 any objects defined within a module must be mentioned in an "EXPORT" statement before the object can be used outside of the module. In a similar fashion, any object to be used inside of module and defined outside of the module must be mentioned in an "IMPORT" statement.
- **Shorthand Expressions** — Modula-2 does not provide any of the shorthand expressions available in C. For example, in C it is possible to use "i++" or "i+=1" as a shorthand for the statement "i = i + 1".
- **Register Binding** — With C a programmer can have direct control over which hardware registers are used to contain frequently accessed program data. In Modula-2 the binding of variables to hardware registers is left to the optimizing section of the compiler.

- Low Level Operations — Part of the Modula-2 language is the facility to specify a start address for a variable when the variable is declared.

For example:

```
exception[0] : ARRAY [divbyzero..trap4] of ADDRESS;
```

- Variable Length Parameter Lists — The C language allows a procedure to be called at different times with a different number of parameters. In Modula-2 a procedure must always be called with the same number of parameters.

## 2. Modula-2 Implementations.

The development of the first Modula-2 compilers was done at the Information Institute of the Federal Institute of Technology in Zürich, Switzerland (Institute für Informatik, Eidgenössische Technische Hochschule, "ETH"). All of this early development was supervised by Niklaus Wirth. These first compilers were all multi-pass compilers written in either PASCAL or Modula-2. Optimized code generators were developed for the PDP-11 family, for the Motorola MC6809 and MC68000, and for the generic Modula-2 machine (m-codes). Since then code generators for the VAX-11 family, the Intel 8086 family, and the Z80 have been developed. Parallel effort by many people has supported the use of these compilers on many widely used operating systems. For some processors and operating systems more than one Modula-2 compiler is now available. Many module libraries are now in the public domain. This widespread development effort is making Modula-2 available and usable in the more common programming environments.

One of the Modula-2 compilers developed at ETH is called Smiler2. The original version of Smiler2 produced S-records for downloading to a MC68000 development system. The Smiler2 compiler was written in PASCAL Release 3, an implementation for the CDC 6600 (a machine with a 60-bit word size using octal notation in its machine language).

A project was started at the beginning of the 1984 summer break to transport the Smiler2 compiler to a MC68000 Unix operating system environment with the goal of providing Modula-2 there as a viable alternative to C. Not only was it necessary to convert Smiler2 to a 32 bit hexadecimal notation machine, it was also necessary to enhance the compiler to support the binary objects files for the use of Modula-2 in a Unix system.

The computer system chosen to support the Smiler2 conversion effort was the Charles River Data System Universe 68 with the Silicon Valley Software PASCAL compiler. Three people have worked part-time on the conversion effort over a six-month time period. Extensive additions were required to support the "a.o" and "a.out" formats of the UNOS operating system for the CRDS MC68000 hardware system. (UNOS is a real-time operating system compatible with Unix System V.) The UNIX/Smiler2 implementation can link to any Unix C subroutine library using the Unix loader (ld). If all of the modules do not require access to C subroutines, the executable image is then produced without the use of the Unix loader. Access to a C subroutine is implemented by importing a procedure from any module with a name beginning with the two letters "C\_".

Recently the UNOS version of Smiler2 was transported to a BSD V4.2 Unix system. This allows the development of Modula-2 programs for

downloading to custom MC68000 hardware applications in the more advanced and traditional VAX-11/Unix environment.

### 3. Evaluation of Modula-2.

To test the usefulness of Modula-2 in a MC68000/UNIX environment, a number of Modula-2 projects were done as validation. The four questions we attempted to answer are given here.

- Can existing Modula-2 programs be easily converted to UNIX?
- Can Modula-2 programs be easily written to access specific UNIX features?
- Can a device control application written in C be replaced by one written in Modula-2?
- Can new device control applications be developed in Modula-2?

#### 3.1. On Unix.

The University of Montana has been using Modula-2 in some of its Computer Science courses for more than two years. As a result it has developed a number of programming tools in Modula-2 which it is using with the RT-11 Modula-2 compiler developed at ETH Zürich. These tools include libraries for file handling, real numbers, strings, a pre-emptive process scheduler and tools to support the monitor concept for concurrent programming. The libraries require access to "read()", "write()", and access to some procedures in the "stdio" library.

The pre-emptive scheduler requires a clock interrupt. The clock interrupt can be simulated on the UNOS operating system with a mechanism called "event-counts". In custom MC68000 systems the pre-emptive scheduler can connect directly to the hardware clock interrupt.

An example of access to the Unix "write()" from a Modula-2 program in the UNOS environment follows:

```
(* Low level procedure taken from SYSTEMX module. *)
(* Jwrite() is equivalent to write(). *)
(*$P- No call frame is needed. *)
PROCEDURE Jwrite(VAR buffer: ARRAY OF CHAR; lu: INTEGER);
BEGIN
  CODE(SAVE, {0..2}, SP);          (* save d0-d2 on stack *)
  CODE(LOAD, (SP+16)*LS9, {0..2}); (* push args into regs *)
  CODE(CLR+LONG*LS6+D3);          (* clear reg d3 *)
  CODE(MOVEW+14*LS9+A0);          (* put call # 14 into A0 *)
  CODE(TRAP+2);                   (* trap 2 calls UNOS *)
  CODE(CMPL+E INTERRUPTED*LS9+D0); (* transfer interrupt? *)
  CODE(BNE+16);                   (* branch if no interrupt *)
  CODE(SUBL+A0*LS9+D1);           (* D1 now 0, bytes written *)
  CODE(ADD+LONG*L6+D1*L9+D2);     (* D2, count not written *)
  CODE(SUB+LONG*L6+D1*L9+D3);     (* restore cumulative byte *)
  CODE(MOVEL+A0*LS9+D1);          (* D1 gets pointer *)
  CODE(MOVEL+(16+SP)*LS9+D0);     (* reload lu into D0 *)
  CODE(BRA, -28);                 (* continue the call *)
  CODE(TST+LONG*LS6+D0);          (* terminated normally? *)
  CODE(BLT+2);                   (* no, return error code *)
```

```

CODE(ADD+LONG*LS6+D3*LS9+D0); (* add in the byte count *)
CODE(LOAD,{0..2},SP);          (* restore the registers *)
CODE(RTS);                      (* return to caller *)
END Jwrite;

```

A sample Modula-2 program which uses the low level Jwrite() to call UNOS follows:

```

IMPLEMENTATION MODULE test;
FROM SYSTEMX IMPORT Jwrite;      (*

WriteLn and WriteString are used to print a variable length
string or a Cr/Lf. Jwrite() is an assembly language routine
which makes a trap call to UNOS. It takes three arguments:
    1. File descriptor (Lu) number.
    2. Pointer to the buffer.
    3. Size of buffer.

Smiler2 passes two arguments for open arrays, one is the
size of the array, the other is the address. Smiler2 passes
the arguments in reverse order. *)

```

```

PROCEDURE WriteLn;
VAR k: ARRAY [1..2] OF CHAR; (* allocate a buffer *)
BEGIN
    k[1] := CHR(13);          (* buffer a CR *)
    k[2] := CHR(10);          (* buffer a LF *)
    Jwrite(k,1);              (* call Jwrite to print *)
END WriteLn;

PROCEDURE WriteString(VAR X:ARRAY OF CHAR);
BEGIN
    Jwrite(X,1);              (* pass the array to Jwrite *)
END WriteString;

BEGIN (* test *)
    WriteString("Hello World!"); (* print a message *)
    WriteLn;
END test.

```

### 3.2. Custom MC68000 Applications.

A two-board raster printer controller for the Motorola VME buss made by Electronic Machines Corporation converts text to raster information. All printing is done without buffering a whole page of raster information prior to printing a new page of paper. This requires complex computation while a page is being printed. When the EMC boards are used in a MC68000 system to control a high speed laser printer, a control program is needed that can respond to five different device interrupts. While these devices are interrupt enabled the control program also needs to compute the rasterization. This application becomes even more complex when the controller becomes a print server node in a network. A Modula-2 program is being developed to control the EMC devices using a concurrent programming design.

The Universe 68 used to transport Smiler2 is based on two MC68000 processors. One MC68000 processor is the main processor running the UNOS operating system. The other MC68000 processor is a dedicated serial data I/O co-processor running a special I/O processing operating system (IOP). The standard IOP provided with the UNOS system is written in C and requires 8K bytes for the program. The exact functionality of the IOP provided with the UNOS system is being re-written in Modula-2 resulting in a program which can be directly compared with the original C language implementation.

An example of a C procedure we converted to Modula-2 follows:

```
/*
(C) Copyright 1984, Charles River Data System Inc.
Check to see if the port is a tty or a parallel port.
If it is a parallel then call the parallel port handler.
If it is ( not (XOFF) and printing is enabled )
    then advance the pointer to the mailbox.
If transmit is OK then
    If the output char is -1 then reset the device.
    else give the output char to the hardware.
*/
txserve(dev)    /* call xmtr service routine for device */
DEV dev;
{
    register SCC *s;
    register PORT *pt;
    register int c;
    if ( ! (dev & 0x80) ) {
        s = &scs[dev];
        pt = &sport[dev] ;
        if (!(pt->p_flags & OSTOP) && pt->p_flags & QWRITE)
            pt->p_wp.writeptr = pt->p_writembx->mb_bufp;
        if(s->s_cr & TXREADY) {
            if (c = getachar(dev)== -1 )
                s->s_data = RESETTXI ;
            else
                s->s_data = c;
        }
    }
    else
        parserve(dev);
}
```

Permission to include the above copyright material in this article has been granted by Charles River Data System Inc.

The preceding procedure might be implemented in Modula-2 as follows:

```

MODULE iopmain;

FROM SYSTEM IMPORT ADDRESS;

TYPE

(* List all the possible serial ports and their states. *)
TTYs = (tty0, tty1, tty2, tty3, tty4, parallel);
DEV = [tty0..parallel];
scctype = ( TXREADY, RESETTXI);

(* structure of the SCC chip *)
SCCPTR = POINTER TO SCC;
SCC = RECORD
    csr : SET OF scctype;    (* control status *)
    data: INTEGER;          (* data buffer *)
END;

VAR SCCS : ARRAY [tty0..parallel] OF SCCPTR;

PROCEDURE txserve(dev:DEV);
VAR
    S : SCCPTR;
    c : INTEGER;
BEGIN (* txserve *)
    IF NOT ( parallel = dev )
    THEN S := SCCS[dev];    (* get pointer to device *)
    IF TXREADY IN S^.csr
    THEN c := getachar(dev);
        IF c = -1
        THEN INCL(S^.csr, RESETTXI) END(* reset port*)
        ELSE S^.data := c END
    ELSE parsev(dev) END;
END txserve;

BEGIN (* iopmain *)
...
END iopmain.

```

#### 4. Future uses of Modula-2.

A number of projects are planned which will use Modula-2 as the implementation language in the UNIX/MC68000 programming environment. It will take some time before the results of these efforts to use Modula-2 are available for study and analysis. Only then will the real usefulness of Modula-2 begin to be understood.

## 5. Summary.

The conversion of Smiler2 to a UNIX environment has enabled the investigation of Modula-2 as an alternative to the C programming language. Conversion of useful libraries and utilities, the production programs for execution on Unix, and the development of control software for downloading to co-processors and printer controllers have demonstrated that Modula-2 can be used as systems programming language in the UNIX programming environment.

The benefits Modula-2 brings to complex system programming by supporting very clear and modular program design is a new software engineering concept for many programmers using PASCAL and C. Some of the recent articles about Modula-2 have attempted to explain modular program design for concurrent applications [Gut84] [Sew84]. In time, some of the successful design approaches will find general recognition in the programming community.

Recent evaluations of Modula-2 have raised concerns about the lack of standard definitions for I/O and Math libraries [Mof84] — a very primitive I/O environment is described as part of the language definition. The language was meant to enable development of libraries specific to a particular application.

Many procedure libraries developed by early Modula-2 users have entered the public domain. Modula-2 programmers are passing these libraries to each other, converting them to new operating systems, all the while making additions and improvements.

Over time a process of natural selection will occur resulting in some initial confusion but a long-term de facto standard for libraries written in Modula-2. The issue of standard I/O and Math libraries is still something to be resolved — perhaps by the growing Modula-2 user community.

It is our contention that our efforts and the efforts of others clearly show the usefulness of Modula-2 in the Unix environment. Modula-2 brings to complex program implementation a concise method of stating the logical organization of a design. The clarity, integration, and flexibility of Modula-2 will increasingly promote its use in programming applications.

## 6. Acknowledgements.

We would like to thank a number of people who have helped us with this investigation of Modula-2. K. Y. Tan for the many long weeks spent converting Smiler2 to SVS Pascal. Bob Mulchay of Charles River Data System for his support and help in obtaining the Universe 68 hardware system. Jeff Barth of Silicon Valley Software for providing his support and the Pascal compiler. Rob Stuehler for his crafty re-wording of some awkward sentences. And John Copeland of ImageNetwork for providing access to typesetting facilities for the preparation of this paper.

## 7. References:

- [Coa84] Coar, D. "Pascal, Ada, and Modula-2". In *BYTE*, p. 215-232, August, 1984.
- [Hop84] Hoppe, J. "Some Problems with the Specification of Standard Modules". In *Modula-2 News*, Issue #0, October, 1984.

- [Gut84] Gutknecht, J. "Tutorial on Modula-2". In *BYTE*, p. 157-176, August, 1984.
- [McC83] McCormack, J. and Gleaves, R. "Modula-2 A Worthy Successor to Pascal". In *BYTE*, p. 385-395, April, 1983.
- [Mof84] Moffat, D.V. "Some Concerns About Modula-2". In *SIGPLAN Notices*, ACM, Vol. 19, Num. 12, p. 41-47, December, 1984.
- [Pau84] Paul, R.J. "An Introduction to Modula-2". In *BYTE*, p. 195-210, August, 1984.
- [Pow84] Powell, M. "Using Modula-2 for System Programming with Unix". In *Proceedings of USENIX Association, Summer Conference*, p. 119-132, 1984.
- [Sew84] Sewry, D. "Modula-2 Process Facilities". In *SIGPLAN Notices*, Vol. 19, Num. 11, p. 23-33, November, 1984.
- [Sew84] Sewry, D. "Modula-2 and the Monitor Concept". In *SIGPLAN Notices*, Vol. 19, Num. 11, p. 33-42, November, 1984.
- [Spe82] Spector, D. "Ambiguities and Insecurities in Modula-2". In *SIGPLAN Notices*, ACM, Vol. 17, Num. 9, p. 43-51, September, 1982.
- [Wir71] Wirth, N. "The Programming language PASCAL". In *Acta Informatica 1*, p. 35-63, 1971.
- [Wir77] Wirth, N. "Modula: a Language for Modular Multiprogramming". In *Software Practice and Experience*, Vol 7, p. 3-35, 1977.
- [Wir80] Wirth, N. "Modula-2". In *Berichte des Instituts für Informatik der ETH Zürich*, Nr. 36, 1980.
- [Wir83] Wirth, N. *Programming in Modula-2*. Springer-Verlag, Second Corrected Edition, 1983.
- [Wir84] Wirth, N. "History and Goals of Modula-2". In *BYTE*, p. 145-152, August, 1984.
- [Wir84] Wirth, N. "Revisions and Amendments to Modula-2". In *Modula-2 News*, Issue #0, October, 1984.

CDC 6600 is a trademark of Control Data Corporation.

8086 is a trademark of Intel.

MC6809 and MC68000 are a trademarks of Motorola Corporation.

PDP-11, RT-11, and VAX-11 are trademarks of Digital Equipment Corporation.

Universe 68 and UNOS are trademarks of Charles River Data System.

Unix and System V are trademarks of AT&T.

Z80 is a trademark of Zilog.

# Concurrent C—An Overview\*

N. H. Gehani

W. D. Roome

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

Concurrent C is a concurrent programming superset of the C programming language. This paper gives a brief overview of Concurrent C, and gives an example of a Concurrent C program. We have implemented Concurrent C on a UNIX<sup>TM</sup> system running on a single processor.

### 1. Introduction

Concurrent programming is becoming increasingly important because multicomputer architectures, particularly networks of microprocessors, are rapidly becoming attractive alternatives to traditional maxicomputers. Concurrent programming is important for many reasons [HOAR78, GEHA84c]:

- Notational convenience and conceptual elegance.
- Expressing concurrent algorithms; otherwise, the structure of the algorithm may be lost.
- Reducing program execution time on genuine multiprocessing hardware.
- Reducing program execution time even on single CPU machines because I/O and the CPU operation can proceed in parallel.

We have added a set of concurrent programming facilities to the C programming language [KERN78, RITC80]; we call the resulting language Concurrent C. This paper gives a brief overview of Concurrent C, and shows how the dining philosopher's problem can be solved in Concurrent C. For additional information on Concurrent C, see [GEHA84a, GEHA84b].

### 2. Selection of the Concurrency Model

Concurrent C is based on a synchronous message passing, or *rendezvous*, model. We selected this model for the following reasons. First, we rejected shared memory models, because we wanted the freedom to implement Concurrent C on a network of computers that do not share memory.<sup>1</sup> That left us with message passing models. These fall into two categories: synchronous (blocking send) and asynchronous (non-blocking send). We choose synchronous message passing because (a) it is well known and has been discussed in the literature [HOAR78, DOD83], (b) the language extensions to support this model are straight-forward, and (c) it can be implemented efficiently.

Synchronous message passing primitives combine process synchronization with information transfer. While asynchronous message passing models have two primitives—*send* and *receive*—synchronous message passing models have three primitives: *send-and-wait-for-reply*, *accept-request*, and *reply*. One process initiates an interaction by using the *send-and-wait* primitive; typically this is a "client" process requesting service from a "server" process. The client process is blocked until the server process *accepts* that request, handles it, and *replies* to the client. After that, the client and server processes proceed in parallel. This interaction is known as an *extended rendezvous* or *transaction* [DOD83]. From the client's viewpoint, a transaction request is just like a function call.

\* This paper is an abbreviated version of *Concurrent C* [GEHA84a] and *Concurrent C—Programming Examples* [GEHA84b], and has been prepared for the 1985 Winter USENIX Conference.

1. We don't object to shared memory; in fact, a distributed implementation of Concurrent C could make very effective use of shared memory. We just did not want to limit ourselves to tightly-coupled, shared memory multiprocessors.

Although Concurrent C and Ada [DOD83] are both based on the rendezvous model, there are important differences between the concurrent programming facilities in the two languages. These differences are summarized in [GEHA84a].

### 3. Concurrent C

A Concurrent C program consists of a set of one or more processes executing in parallel and cooperating with each other to accomplish a common goal. We will now describe the facilities in Concurrent C.

#### 3.1 Processes and Process Types

First, some terminology. A *process type* is a process template. A *process* is an instantiation of a process type. A process has its own flow-of-control; it executes in parallel with other processes. Declaring a process type does not automatically create a process of that type. Instead, the programmer must explicitly create each process at run-time.

A process type has two parts: a *specification* and a *body*. The process type specification (or process specification, for short) is the public part of a process type. Only the information specified in the process specification is visible to other processes. A process specification gives all the information necessary for interacting with a process of that type, or for creating a new process of that type. A process body contains the code (and associated declarations and definitions) that is executed by a process of this type; it is analogous to a function body, which it resembles. Details of the process body are not visible to other processes.

#### 3.2 Process Specifications and Transaction Declarations

A process specification has the general form

```
process spec process-type-name( types of process parameters )
    { transaction declarations };
```

A transaction declaration has the form

```
trans return-value-type transaction-name( formal parameter types );
```

As an example, the following process specification says that process type `buffer` has one parameter (`int`), and two transactions, named `get` and `put`:

```
process spec buffer(int)
{
    trans void put(char);
    trans char get();
};
```

Concurrent C processes communicate primarily by means of transactions. The process containing a declaration for a transaction named `T` may be thought of as a server for transaction `T` because it will execute requests for this transaction on behalf of the client processes. (Note that a process may act as a server for more than one transaction). The formal parameters represent the information that a client gives to the server; the return type is the information that the server returns to the client.

#### 3.3 Process Bodies

A process body has the form:

```
process body process-type( names of process parameters )
    compound statement
```

Each process is a sequential program component that runs independently and in parallel with other processes. Process parameters can be used in the process body just as function parameters are used in function bodies. Values for the process parameters are supplied when creating a process. As an example, here is a skeletal body for the process type `buffer`:

```

process body buffer(max)
{
    char *bufp;

    bufp = malloc(max);
    accept get and put transaction requests
    free(bufp);
}

```

A process body cannot reference global variables; this is how we discourage the use of shared memory. However, a process body can call functions that access, update, or return global variables; this is our concession to reality.

### 3.4 Process Creation, Process-Valued Expressions, and Process Variables

The `create` operator creates (instantiates) a new process of the specified type, and specifies the values for the process parameters. For example, the expression

```
... create buffer(128) ...
```

creates a new process of type `buffer`, with 128 for the value of its parameter. The `create` operator returns a value of the process type being instantiated, which in this case is a value of type `process buffer`. A process value identifies a specific process. A process value can be stored in a process variable of the same type:

```

process buffer b;

b = create buffer(128);

```

Process variables can appear in structures, and Concurrent C programs can have arrays of process variables, or pointers to process variables, as in:

```
process buffer bufarr[10], *pb;
```

The reserved name `c_nullpid` is a null (invalid) process value. `c_nullpid` is polymorphic, in that it can be assigned to a process variable of any type.

The general form of the `create` operator is:

```
create process-type-name(process parameter values) [with priority(p)]
```

where  $p$  is an integer expression that specifies the new process's priority relative to some standard priority. If the `create` operator executes successfully, then it returns a process value identifying the created process; otherwise, it returns `c_nullpid`. The created process becomes eligible for execution immediately. The created process is called a *child* process of the creating, or *parent*, process. Processes that have the same parent process are called *sibling* processes.

### 3.5 Process States and Process Termination

A process can be in one of the following three states:

active	A process becomes <i>active</i> upon creation.
completed	A process becomes <i>completed</i> when it executes a <code>return</code> statement in its process body, or when it reaches the end of its body.
terminated	A process becomes <i>terminated</i> when it has completed and all the processes created by it have terminated, or when it executes a <i>ready-to-terminate</i> alternative in a <code>select</code> statement.

Note that a process can *complete* whenever it wants to, but a process cannot *terminate* until all its children (and all their children, and so on) have themselves terminated. A completed but not terminated process is in limbo: it cannot be called, nor can it call other processes.

The `c_abort` function terminates a process and all its children, regardless of the current state of the process or of its children. Processes should therefore be aborted with care.

### 3.6 Transaction Calls

A transaction call is the caller's (i.e., client's) side of a transaction. The syntax is similar to a function call:

*process-value.transaction-name( actual parameters )*

*Process-value* is a process-valued expression. The specification for that process type must have a transaction named *transaction-name*. The transaction call has the type returned by the transaction. In general, a transaction call can be used wherever an expression of that type is allowed.

The calling process is delayed until the called process accepts the transaction (see the `accept` statement, below). The called process is given the values for transaction parameters specified by the caller. The calling process then remains suspended until the called process returns a value; this becomes the value of the transaction call expression. Note that the transaction call is directed to a *specific* process, not to an arbitrary process of that type.

### 3.7 Accept Statements

The `accept` statement is the receiving process's side (i.e., the server's side) of a transaction. An `accept` statement has the form

*accept transaction-name(formal parameter names) [ by ae ]  
compound statement*

where *ae* is an arithmetic expression that should involve the formal parameters of the `accept` statement. An `accept` statement can only appear in the body of a process.

If a process has one or more pending transaction calls for the specified transaction, then the `accept` statement accepts one of them immediately. If the `by` clause is omitted, transaction calls are accepted in first-in-first-out (FIFO) order. If there is a `by` clause, then the arithmetic expression *ae* is evaluated using the argument values in each of the pending calls, and the pending call with the minimum value is accepted [ANDR81]. If there are no pending transaction calls, execution of the `accept` statement is delayed until a transaction call arrives.

Once a transaction call has been accepted, the process executes the compound statement that forms the body of the `accept` statement. Within that compound statement, the formal parameter names become variables that hold the parameter values given by the caller. The types of these variables are those given in the transaction declaration, in the process's specification.

The calling process is delayed until the `accept` statement terminates by executing a `treturn` statement:

*treturn expression;*

The value of the expression is returned to the calling process. The process containing the `accept` statement goes on to execute the next statement (after the body of the `accept`) and the process issuing the transaction call becomes free to resume execution. Note that the reply is asynchronous; the `treturn` statement does *not* wait for the calling process to resume.

If the transaction is declared as returning a value of type `void`, the `accept` statement terminates by using a `treturn` statement with no value, or by just running off the end of the compound statement.

### 3.8 Select Statements

The `select` statement allows a process to non-deterministically wait for the first of several events. The syntax is:

```

select {
  [(guard1):] alternative1
or
  [(guard2):] alternative2
or
  :
or
  [(guardn):] alternativen
[otherwise statements]
}

```

A *guard* is a boolean expression (non-zero is true and 0 is false). The order in which guards will be evaluated is unspecified. Thus we suggest that guards be kept as simple as possible, and that side-effects in guards be avoided.

A *select statement alternative* can be one of the following, depending on the first statement in the alternative:

1. an *accept* statement, optionally followed by other statements (an *accept alternative*),
2. a *delay* statement, optionally followed by other statements (a *delay alternative*),
3. a *terminate* statement (a *ready-to-terminate alternative*), or
4. a *null* statement, followed by other statements (an *immediate alternative*)

The *accept* and *delay* statements were described earlier. The *null* statement is just a semicolon. The *terminate* statement is just:

```
terminate;
```

The *select* statement executes one and only one of the alternatives. Once chosen, all statements in that alternative are executed, and execution resumes after the *select* statement. The following rules determine which alternative will be taken:

1. If there is an *accept* alternative with a true guard (non-zero or omitted), and if a transaction call is waiting for that transaction, take that *accept* alternative.
2. Otherwise, if there is an *immediate* alternative with a true guard, take that alternative.
3. Otherwise, if there is a *ready-to-terminate* alternative with a true guard, and if the termination criteria are satisfied (see below), then take the *terminate* alternative. This terminates the process.
4. Otherwise, if there is an *otherwise* clause, take it.
5. Otherwise, let  $x$  be the lowest delay specified by a *delay* alternative with a true guard, or infinity if there are no such *delay* alternatives. Then if one of the following events occurs within the next  $x$  seconds, then take the corresponding alternative:
  - a. The arrival of a transaction call for one of the transactions for which there is an *accept* alternative with a true guard.
  - b. The occurrence of the *termination conditions* (discussed below), if there is a *ready-to-terminate* alternative with a true guard.

If none of these events occur within  $x$  seconds, take the *delay* alternative.

It is considered to be an error if none of the guards are true and there is no *otherwise* clause.

The *termination conditions* mentioned above are true if all of the following conditions are true:

- All the children of this process have terminated.
- All sibling processes (other processes created by the parent process) have terminated, or are waiting at a *ready-to-terminate* alternative and their children have terminated.

- The parent process has completed execution.<sup>2</sup>
- This process has no pending transaction calls.

### 3.9 Timed Transaction Calls

A timed transaction call allows the process requesting the transaction to withdraw the transaction call if the transaction is not accepted within a specified period. A timed transaction call is an expression of the form

*within duration ? p.t(actual parameters) : expression*

where *duration* is floating point expression, *p* is a process-valued expression, and *t* is a transaction name. If process *p* accepts this request within *duration* seconds, then the value of the expression is that returned by the transaction; otherwise, its value is *expression*.

### 3.10 Delay Statements

A process can delay itself by executing a statement of the form

*delay duration;*

where *duration* is a floating-point expression that specifies the amount of the delay in seconds. The actual delay may be more, but not less, than the requested delay.

### 3.11 Some Concurrent C Functions

<i>c_count(t)</i>	<i>c_count</i> returns the number of pending requests for the transaction named <i>t</i> . <i>c_count</i> can only be used within the body of a process.
<i>c_active(p)</i>	<i>c_active</i> returns 1 if process <i>p</i> is active, or 0 if not.
<i>c_completed(p)</i>	<i>c_completed</i> returns 1 if process <i>p</i> has completed, or 0 if not.
<i>c_invalid(p)</i>	<i>c_invalid</i> returns 1 if <i>p</i> refers to a terminated or an invalid process, or 0 if not.

## 4. A Concurrent C Programming Example—The Mortal Dining Philosophers

Here is the obligatory dining philosophers problem [GEHA84c]:

Five philosophers spend their lives eating spaghetti and thinking. They eat at a circular table in a dining room. The table has five chairs around it and chair number 1 has been assigned to philosopher number 1 ( $0 \leq i \leq 4$ ). Five forks have also been laid out on the table so that there is precisely one fork between every adjacent two chairs. Consequently there is one fork to the left of each chair and one to its right. Fork number 1 is to the left of chair number 1.

Before eating, a philosopher must enter the dining room and sit in the chair assigned to her. A philosopher must have two forks to eat (the forks placed to the left and right of every chair). If the philosopher cannot get two forks immediately, then she must wait until she can get them. The forks are picked up one at a time. When a philosopher is finished eating (after a finite amount of time), she puts the forks down and leaves the room.

The dining philosophers problem has been studied extensively in the computer science literature. It is used as a benchmark to check the appropriateness of concurrent programming facilities and of proof techniques for concurrent programs. It is interesting because, despite its apparent simplicity, it illustrates many of the problems, such as shared resources and deadlock, encountered in concurrent programming. The forks are the resources shared by the philosophers who represent the concurrent processes.

2. Without this condition, a process could terminate before its parent or other siblings can interact with it.

The five philosophers and the five forks will be implemented as processes. On activation, each philosopher is given an identification number (0-4) and the process values of the forks she is supposed to use. Each philosopher is mortal and passes on to the next world soon after having eaten 100,000 times (about three times a day for 90 years).

The specifications of the philosopher and fork processes are:

```
process spec fork()
{   trans void pick_up();
    trans void put_down();
};

process spec philosopher(int, process fork, process fork);
```

The bodies of the philosopher, fork, and main processes are:

```
process body philosopher(i, left, right)
{   int times_eaten;

    for (times_eaten = 0; times_eaten != 100000; times_eaten++) {
        /*think for a while; then enter dining room and sit down*/
        /*pick up forks*/
        right.pick_up(); left.pick_up();
        /*eat*/
        printf("Philosopher %d: That was delicious!\n", i);
        /*put down forks*/
        left.put_down(); right.put_down();
        /*get up and leave dining room*/
    }
    printf("Philosopher %d: See you in the next world\n", i);
}

process body fork()
{   for (;;)
        select {
            accept pick_up();
            accept put_down();
        }
    or
        terminate;
}

main()
{   process fork f[5];
    process philosopher p[5];
    int j;    /*loop variable*/

    /*create the forks and philosophers; the forks must be*/
    /*created before the philosophers*/
    for (j=0; j<=4; j++)
        f[j] = create fork();
    for (j=0; j<=4; j++)
        p[j] = create philosopher(j, f[j], f[(j+1)%5]);
}
```

Once the philosophers have terminated, the forks have nothing else to do; because each fork is waiting at a `select` statement with a `terminate` alternative, and because the parent process (i.e., `main`) has completed, they all terminate. This allows the `main` process to terminate, which completes execution of the Concurrent C program.

## 5. Conclusions

The concurrency model in Concurrent C is based on the rendezvous concept. Many of the differences between the Concurrent C and Ada implementations of the rendezvous are a result of trying to eliminate the polling bias in Ada [GEHA84d], to increase programmer control of scheduling, and to provide the programmer with more control of the concurrency (process parameterization and explicit process activation).

Concurrent C can be used for a variety of applications:

1. To implement parallel algorithms, such as simulation programs and protocols.
2. To write genuinely distributed applications, such as distributed databases.
3. To write real-time programs.
4. To write dedicated programs that execute on a "bare" machine.
5. To implement operating systems.

We have implemented Concurrent C on a UNIX<sup>TM</sup> system [UNIX82, UNIX83] running on a single processor. We are now planning to implement a distributed version of Concurrent C that will allow a Concurrent C program to run on multiple processors.

## References

- ANDR81 Andrews, G. R. Synchronizing Resources. *TOPLAS*, v3, no. 4, 1981.
- DOD83 *Reference Manual for the Ada Programming Language*. United States Department of Defense, January 1983.
- GEHA84a Gehani, N. H. and W. D. Roome. Concurrent C. AT&T Bell Labs, 1984.
- GEHA84b Gehani, N. H. and W. D. Roome. Concurrent C—Programming Examples. AT&T Bell Labs, 1984.
- GEHA84c Gehani, Narain. *Ada: Concurrent Programming*. Prentice-Hall, 1984.
- GEHA84d Gehani, N. H. and T. A. Cargill. Concurrent Programming in the Ada Language: The Polling Bias. *Software—Practice and Experience*, v14, no. 5, pp. 413-427.
- HOAR78 Hoare, C. A. R. Communicating Sequential Processes. *CACM*, v21, no. 8, pp. 666-677, August 1978.
- KERN78 Kernighan, B. W. and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- RITC80 D. M. Ritchie. *The C Programming Language—Reference Manual*. Bell Labs, September 1980.
- UNIX82 *UNIX System User's Manual (Release 5.0)*. Bell Laboratories, June 1982.
- UNIX83 *UNIX Programmer's Manual (4.2 BSD)*. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, August, 1983.

**ABSTRACT**

The symbiosis that makes the UNIX<sup>1</sup> system a friendly and efficient environment for C programs can present problems for a language having an execution model much different from C's. The problems are compounded if C functions are to be called from the other language; C's view of the execution environment must be preserved.

Our UNIX-based Ada<sup>2</sup> runtime system provides a convenient interface to C libraries, including UNIX system calls, while correctly implementing required Ada features and automatic storage reclamation. As designers and implementors, we have reconciled conflicting requirements and assumptions about main program startup, subroutine linkage, data representation, and other subjects. We discuss three major design problems and our solutions:

*Differences in the UNIX process and Ada task models:* UNIX interprocess communication assumes a flat data space, but Ada tasks have nested name spaces. Processes and tasks have different models of dependency and termination. Ada requires task scheduling services that are different from those provided to UNIX system processes. UNIX processes are larger, more complex entities than many Ada tasks. To run each Ada task as a separate UNIX process would constrain the number of tasks in a program and place a burden on other user programs on the system.

Our solution to the problems arising from the process-task mismatch is to run an Ada program as a single UNIX process, irrespective of the number of tasks spawned by the program. We provide inter-task communication, termination, and scheduling facilities in our runtime kernel. A non-blocking *read()* system call would enhance the performance of multitasking Ada programs.

*Handling signals in Ada:* UNIX system signals do not distinguish between synchronous *traps* caused by program execution and asynchronous *interrupts* arising from external events. While this is convenient and appropriate in a time-sharing environment, Ada's primary application domain is real-time embedded systems; a distinction is made between synchronous *exceptions* and asynchronous interrupts. We simulate the required Ada semantics by mapping the set of signals that can reasonably arise through program execution to exceptions and consider others to be external interrupts.

*Dynamic storage allocation:* Ada programs can allocate arbitrarily complex heap structures. The same runtime kernel facilities that manage the heap provide multiple stack segments within the single process space for multitasking programs. We provide safe and complete storage reclamation for heap objects, reclaim storage incrementally when convenient, and impose the cost of full storage reclamation only on those programs that use dynamic storage heavily. Existing C libraries are callable from Ada programs; C functions that use dynamic storage may link to and call a new *malloc()* family interface that interacts with our heap allocator in a way that is transparent to the C functions.

---

1. UNIX is a trademark of AT&T Bell Laboratories.

2. Ada is a registered trademark of the U.S. Government — Ada Joint Program Office.

## Development of a Compiler for the Bourne Shell

Vince Kasten

Paul Ruel

Concentric Associates, Inc.

shacc is a compiler for the Bourne shell. It translates Bourne shell scripts into C code which is compiled into a final a.out file. The paper discusses the specific design goals of the project, the complexity of the problem, the general approach to the resulting software and preliminary assessment of the performance of the final product.

The design goals discussed are

- Speed and resource utilization efficiency
- Completeness of the implementation
- Portability and appearance of the resulting C code
- Facilities for language extension in later releases

Compiling a language designed for interpretation is rarely easy. Irregularities in both the syntax and semantics of the Bourne shell presented a number of special case grammatical problems. These are discussed in principle, with specific (but not exhaustive) examples of troublesome ambiguities and inconsistencies.

Due to the many special cases, generation of the lexical scanner and the grammar was something of an adventure -- the tendency of the lex-generated code to grow exponentially forced hand-coding of the scanner. This is discussed in terms of those constructs that caused problems. The basic approach to code generation, and the data structures used to generate the code, are discussed.

## 1. Why Compile the Shell?

Aside from being one of the two most common command language interfaces to UNIX\* systems, the Bourne shell is widely used as a programming language. The UNIX system software is a combination of shell code along with C code, individual users write shell programs to perform specialty functions, and many software developers use shell code to prototype pieces of software systems. The advantages of using shell code when the application is appropriate, are obvious. However there are several cases where shell scripts are better replaced by a.out executables.

If a program is executed repeatedly during the day, it is advisable to set the sticky bit to save the text image. This minimizes the amount of filesystem I/O traffic generated by the execution of the program. The sticky bit only works with a.out files. Occasionally a program must be written that uses the setuid file mode. This only works with a.out files.

Often shell tools are used to support a software development effort. During development, if these programs are likely to change, they are probably best left as shell scripts. If they are stable, recoding them in C will probably make them less of a drag on system resources. Moreover, we have seen many software projects wind down with shell "tools" remaining as part of the deliverables. Often these shell scripts reveal more of the implementation details of the product than one would like. Thus an eleventh hour recoding effort is necessary.

This paper discusses the development of a compiler for the Bourne shell: shacc. In fact, shacc is not a formal compiler. It is a program that translates Bourne shell scripts into C code. The system's native compiler is used to produce the a.out.

## 2. Design Goals For shacc

The specific design goals for shacc were

- The ability to translate as many Bourne shell constructs as possible
- The generated code should be well-structured, readable and portable
- The implementation should allow for later extensions to the Bourne shell language

Initially, the shacc design goal was to implement a compiler for the complete Bourne shell. Some initial investigation uncovered two Bourne shell constructs that are virtually impossible to compile — notably the "eval" and the "." command. Both of these are essentially language features that allow self-modifying code. As development proceeded, another surprise developed in the here document.

We felt it was important that the compiled code be well-structured and readable, and essential that the code be portable. In truth, the structure and readability requirements were an aesthetic issue; we did not want code like that generated by yacc, and the goal is perfectly reasonable. Later when we found constructs that we couldn't reasonably compile, readability was essential. We felt that just because shacc couldn't handle a construct was no reason not to compile the rest of the program and make it available for hand modification.

The facility for extension is also very important. First, the shell language can change — witness the ksh extension to the Bourne shell. Second, the availability of building C programs from shell code will make it attractive to directly call C and assembly language functions from a shacc'd shell script. Third, as development proceeds to the next release, more commands will be built in to a shacc library, extending the shell builtin set.

---

\* UNIX is a trademark of AT&T Bell Laboratories

### 3. The Scanner and Parser

shacc is a two-pass processor. In the first pass, lexical analysis and parsing, the entire shell script is scanned, building the internal data structures necessary for code generation. The initial development was based on `lex` and `yacc`. We abandoned `lex` early in the development, since it could not handle the load. `yacc` has served well for the grammar.

#### 3.1 The Scanner

The scanner's function is conceptually straightforward. It breaks the input lines into words delimited by those things that the shell recognizes as word/line delimiters. In the shell, recognizing a word is complicated by the environment variable `IFS`, and by double and single quotes. Treatment of `IFS` is a run-time issue that manifests itself at code generation time. We will discuss it in Section 4. The discussion of quoting follows.

Paired single quotes remove the special meaning of all enclosed characters. Thus the scanner simply recognizes a single quote and reads along until it encounters another single quote, internally quoting every enclosed character. Then it returns `WORD` to the parser, along with all of the individually quoted characters. Now look at this perfectly reasonable construct:

```
cd '/usr'/bin'
```

The scanner returns two `WORDS` to the parser, but in reality there is only a single word. The solution is simple enough — just indicate to the parser that there are two `WORDS` that must be glued together before use.

A nastier problem arises with this lexically inconsistent use of single quotes in the `trap` command. Consider

```
trap 'rm /tmp/xx$a;exit 1' 0 1 2
```

Here the standard single quote strategy breaks down. The `WORDS` inside the quotes must be recognized by the scanner, or else either the parser will be forced to rescan

```
trap WORD 0 1 2
```

or the rescan of the line will have to happen at run-time, neither of which is acceptable, since `shacc` must be capable of generating the appropriate `"signal()"` code when it generates code (implying that interpretation of the `trap` string occurs at translation time instead of run-time). All of this means that the scanner must be made aware that it is reading a single-quoted string that is the argument of a `trap` command. This is a pathological connection between the scanner and the parser that seems unavoidable.

Double quotes are a different problem. Double quotes remove the special meaning of almost everything between them, except parameter and command substitution and the meaning of the backslash. Thus the treatment of double quotes is different in the scanner than the treatment of single quotes. A double quoted string gets recognized as a `WORD`, but without the flags that mark each character as quoted. This is important since code must later be generated that will allow run-time interpretation of parameters and command substitutions.

All other constructs are straightforward, with the scanner aware of the special meaning of the list separators and the `I/O` redirection symbols.

As is easily seen, quoting is a mass of special cases. These all translated to extra start states in the initial `lex` specification. As advertised in its documentation, `lex` dutifully grew exponentially, eventually to 64K. At that point it was thrown away and the lexical scanner was recoded in C. It is now approximately 10K.

### 3.2 The Parser

The parser is written using yacc, and it is assumed that the reader is familiar with yacc grammars. Given this, we won't examine any of the parser specification. It is a standard grammar, but greatly complicated by the multitude of special cases encountered.

In some sense, interpreters are easier to write than compilers. If an interpreter is confused, it just reads more characters until it gets unconfused or until it dies. As a result, interpreters tend to have "features" beyond those intended by their designer. As long as the interpreter does the right thing for the language definition, other cases are left to default (we will see obvious cases of this as we examine some surprises later in this section).

With shacc we didn't have the luxury of being so cavalier. First, shacc must recognize things unambiguously at compile time so it can generate code that will run. Second, since shacc is a translator for a language that has existed for seven years it must perform translation consistent with the ultimate language definition available, the shell itself. Thus, this section is really a look at some of the nastier undocumented "features" of the Bourne shell and why they are a problem.

The simple command list is the logical place to start. It appears innocuous enough in sh(1):

```
{list;} is simply executed
```

First of all, the manual is incorrect, because the "{" requires that white space separate it from the first command in the list. Unless list is in a subshell. That is

```
{list;}
```

is no good, but

```
{(list)}
```

is acceptable, with no white space between "{" and "(" and also no ";" terminator.

More fun with the "{" and ";":

```
for i in list
{
    command
    command
}
```

is perfectly acceptable as is

```
case $i {
    pattern) list ;;
    pattern) list ;;
    .
    .
    .
}
```

The case statement itself is difficult given the other uses the shell makes of parenthesis. The right parentheses in the case statement was a contributing reason that the lex grew so large. Also, the case statement allows a unique usage of the pipe symbol, and causes filename generation characters to have different meanings than normal.

Two last pieces of fun in the case statement. The statement

```

case $i in
    esac(pattern) list ;;
esac

```

is illegal, but

```

case $i in
    pattern|esac) list ;;
esac

```

is acceptable. This is an example of "default" behavior in the interpreter that shacc must mimic. Also

```

case $i in
    albl....l) list ;;
esac

```

is acceptable, with the "|" requiring no quote. Unquoted "|" also works in a pattern, but unquoted "(" does not.

One final special case that leads nicely into the section on generating the code has to do with pipelines. Disregarding the semantics of this statement,

```

command | ;

```

is perfectly acceptable to the shell. It *does* generate a mysterious single newline before returning a shell prompt.

We personally feel that most of the things that are discussed in this section are unresolved bugs that manifest themselves as either features or at worst as innocuous special cases. However we felt obligated to implement all of them in a manner consistent with the action of the shell, since somewhere there is someone who has a shell program that depends on one of these. It should be emphasized that the small number of examples presented here are not exhaustive. Literally dozens of such examples exist — we just presented the good ones.

#### 4. Generating the Code

As mentioned previously, shacc is a two-pass translator. The first pass is the scanner and parser. It passes over the shell script, generating a linked list of headnodes that are traversed in the second pass. Each node on this list reflects what type of construct caused it to be generated, and contains threads to other nodes. For example, a shell program

```

echo hello
if
    grep XXX file > /dev/null 2>&1
then
    echo Found XXX
else
    echo "Didn't found XXX"
fi

```

Would generate two headnodes: Simple and ITE (ifthenelse). Simple would be at the beginning, with a link to ITE. Simple would contain threads to the component parts. ITE would also contain threads to the component parts, in this case the IfPart, the ThenPart and the ElsePart. The IfPart would be a Simple, with the Simple node having the information that I/O redirection is taking place. And so on.

The second pass then traverses this list of headnodes, at each node traversing its threads and generating the code for each construct. At appropriate points, code is generated to perform the command line processing that the shell does when interpreting. Thus, as appropriate, code is generated that, at run

time, will perform command substitution, parameter substitution, blank interpretation and filename generation, and then execute the final command. Currently, most simple commands are `fork()`'d and `exec()`'d. For some examples of `shacc`-generated code, see the Appendix.

It is at this stage that some of the major problems with compiling an interpreted language are encountered. Consider the following shell script:

```
X="I work!"
echo $X
. modify
echo $X
```

where the file `modify` has the line

```
X="Not so fast there!"
```

This is impossible to process at compile time, since the file `modify` could easily change. The only solution that presents itself currently is to restrict the use of the `"."` command to be an analog of the `"#include"` facility, i.e.,

```
. modify
```

will cause `shacc` to include `modify` inline at compile time.

Another difficulty arises with the `eval` command. `eval` requires two separate passes over a command line, once to perform substitutions on the line and then once to execute the line performing still more substitutions. Since `shacc` makes one pass at compile time and then executes at run time, implementing the `eval` command would require including code that would run-time-interpret the resulting command line. While this would also give a general solution to the `"."` command problem, it is not considered to be a reasonable solution.

As a final example of something that can not reasonably be compiled, consider this construct

```
cat <<$VARIABLE
where
in
heaven's
name
is
the
eofstring?
```

The only reasonable `shacc` response is to disallow this type of construct, since at translation time it has no way of finding the `eofstring`.

The last look at code generation will concern WORDs in the command line to be executed. Recall that single quoted and double quoted strings are marked in the scanner and passed to the parser. They must be processed at code generation time. Consider the command line

```
echo "$X" 'word'
```

Any quoted string is a single WORD and the two adjacent strings are *still* a single word. But double quoted strings are subject to command and parameter substitution, and premature gluing would yield the word

```
$Xword
```

which would interfere with correct parameter substitution. Thus, these two WORDs must be glued together *after* the command substitution, parameter substitution and blank interpretation phases. This is performed by the `join()` function. `join()`'s are included inline in the generated code as appropriate.

Their only job is to make glued WORDs into one WORD, preparatory to command execution. Obviously the number of words on the command line are not important for the echo command in the example, but are important for commands that count their arguments, such as `cd`.

The converse to `join()` is `split()`. This function is used to perform blank interpretation according to any additions to IFS. This obviously must be performed at run time during the blank interpretation phase of command line processing.

## 5. Future Extensions

The most obvious extension to `shacc` is the addition of built in versions of often-used section 1 commands. This effort is underway; a little thought will show that it is neither straightforward nor easy.

Another extension that is planned is to add the capability to *directly* include C subroutines in the shell script to be `shacc'd`. In essence, this would make `shacc` a superset of the Bourne shell. This is an attractive possibility, but we have yet to work out its exact syntax and semantics.

## 6. Acknowledgements

Paul Ruel of Concentric Associates is the primary developer of `shacc`, with support from the Concentric technical staff.

## 7. Availability

The first release of `shacc` becomes available February, 1985. For licensing information, contact Jim Butz at Concentric Associates, 1 Harmon Plaza, Secaucus, New Jersey, (201) 866-2880.

# Access – A Program to Interpret Pathname Access Permissions for the UNIX Operating System

Stephen J. Mahler

David A. Curry

Engineering Computer Network

School of Electrical Engineering

Purdue University

West Lafayette, Indiana 47907

## ABSTRACT

The program described in this paper, **access**, was written to help users interpret the permissions set on their files by showing in a concise format who may read, write, execute, and remove a given file. Rather than simply reporting the permissions on the file itself, **access** takes into account the permission bits set on the directories which lead to the file. This results in a correct listing of exactly who may access the file, whereas reporting the permissions on the file alone does not.

## Introduction

When a user is first introduced to UNIX he is taught about files – how to edit them, how to read them, how to print them, and how to delete them. Unfortunately, he is rarely instructed on how to protect or control access to them. At the Purdue Engineering Computer Network (ECN), because a large number of our users are faculty and students who use the computer for classwork, file protection is an important aspect of UNIX which we feel must be understood by even the most casual user. Some class instructors even impose a penalty if a student leaves his class files in an unprotected state.

Many users have problems in understanding the file protection facilities under the UNIX operating system. The lack of understanding about the file protection mechanism can be attributed to one or more of the following:

There are three sets of permissions for each file. These are the permissions for the file's owner, other members of the owner's group, and "the rest of the world." Most users would have very little trouble understanding that, "I have one set of permissions, and everyone else has another set." Unfortunately, the concept of groups under UNIX throws a monkey-wrench into the works.

Often new users are all placed into the same default group, the above sentence now becomes, "I have one set of permissions, other people in my group have another set of permissions, and the rest of the world has a third set of permissions. But, since almost everyone else is in my group, then the group permissions are really more like the world permissions, except that those people who aren't in my group still use the real world permissions." It is no wonder that many users are confused.

The program to change file permissions, **chmod**(1), was not written with the novice user in mind. Modes are specified to **chmod** by giving it an octal representation of the permission bits to be set on the file. This may make sense to those who know how UNIX stores file permissions, however, to the average user these are just numbers with no inherent meaning. (This may be due to the confusing nature (to the non-programmer) of bit-fields and the logical operations **and** and **or**.) The newer versions of **chmod** have included a symbolic notation for specifying the modes to be set. This is done by indicating which set (owner, group, world) of permissions to change, and which permissions (read, write, execute) to turn on and off. For example, the command

```
chmod g-r filename
```

would be used to turn off group read permission

on the file *filename*. Although this notation is somewhat unwieldy at times, it is an improvement over the octal notation.

Users often do not realize that file access permissions are affected by the permissions on the directories leading to the file. For example, if a user makes his file readable by everyone, and yet his directory is searchable only by him, he does not understand why his friend cannot copy the file.

In order to help new users protect their files, the ECN has created, over the past several years, four programs which attempt to eliminate the sources of confusion described above. This document is primarily intended to describe the newest and most unique of these programs, **access**. However, because **access** can use the other three programs, they will be described in brief.

### LOCK and UNLOCK

Several years ago, the ECN created two programs for changing file permissions. These programs are called **lock** and **unlock**. The **lock** program removes read, write, and execute permission for the group and world on each file named in its arguments. The **unlock** program restores read and execute permission for the group and world on each file named in its arguments. Both programs, if no arguments are given to them, will change the permissions on the user's current working directory.

Actually, **lock** and **unlock** were implemented as a single shell script, shown below:

```
PATH=/bin
mode=go-rwx

case $0 in
  unlock|*/unlock) mode=go+rx
esac

for name in $(ls -l)
do
  if test ! \(-f $name -o -d $name\)
  then
    echo Making directory $name
    mkdir $name
  fi
  /bin/chmod $mode $name
done
```

### The INFO Program

The **info** program is used to print information about a file. Essentially, it is a combination of the **file(1)** program, and the **stat(2)** system call. For each file named in its arguments, **info** will print the name of the file, its type (directory, normal file, character special file, etc.), its size, a guess at the contents of the file, the owner and group of the file, and the creation, last modification, and last access times for the file. If the **-v** (verbose) flag is given, the number of links to the file, its inode and device numbers are also printed.

An important feature of **info** with respect to file permissions is that it "decodes" the permissions which have been set on the file. These permissions are printed in three formats. First, the octal representation of the permission bits is printed. Second, the "rwx" notation as used by **ls(1)** is printed, and third, a list of which permissions the owner, group, and world have on that file (N.B.: these are permissions on the file only, the directories leading to the file are not considered). This is demonstrated in the example below:

```
$ info /etc/passwd
File Name      - /etc/passwd
File Type      - normal file
File Size      - 115525 bytes, 113 Kbytes
File Contents  - ASCII text
Uid of Owner   - root (0)
Gid of Owner   - root (0)
File Mode      - rw-r--r-- (644)
User root: readable, writable
Group root: readable
Everyone else: readable
Links to file  - 1
Inode Number   - 2130
Inode's Device - 4
Creation Time  - Tue Sep 25 09:12:08 1984
Last Modified  - Tue Sep 25 09:12:07 1984
Last Accessed  - Tue Sep 25 09:55:20 1984
```

### The ACCESS Program

The **access** program is used to actually determine who may read, write, execute, and remove a given file. It is the newest of the four programs, and we believe that it is unique in its approach to interpreting UNIX file permissions. **Access** operates in one of two modes. It can accept command-line arguments, or, if no arguments are given, it enters an interactive mode. When arguments are given, **access** will print out

the access permissions for each file named.

Access determines file access permissions using the following algorithm (see the next section for a more detailed description):

```
file = full path to file
path = "/"
```

**repeat**

*determine access permissions on path*

**for** *i* = 1 **to** MAXUSERS **do**

*determine whether user<sub>i</sub> may read path*

*determine whether user<sub>i</sub> may write path*

*determine whether user<sub>i</sub> may execute path*

*combine this information with that determined for the previous value of path*

**end**

path = path + nextcomponent(file)

**until** path == file

When the algorithm has finished, each user's access permissions for the file in question will be known. This information is then processed in order to determine the most concise way to print it, and the result is then printed on the standard output. For example, the command

```
access /etc/passwd
```

produces the output:

```
/etc/passwd (file):
```

```
Readable by: everybody
```

```
Writable by: root
```

```
Executable by: nobody
```

```
Removable by: root and members of group root
```

If the current directory is */e/davy/system/miscellaneous*, the command

```
access foo
```

might produce the output:

```
/e/davy/system/miscellaneous/foo (file):
```

```
Readable by: members of group nightowl
```

```
Writable by: davy
```

```
Executable by: nobody
```

```
Removable by: davy
```

Note that the entire path to the file is printed.

If **access** is invoked with no arguments, interactive mode is initiated. In this mode, **access** behaves as a special shell, and permits the user to execute various commands to examine and change the permissions on his files. The following

section describes the commands available in the interactive mode.

## ACCESS Interactive User's Manual

When **access** is first invoked, it determines the current working directory. Once this has been done, the prompt *Command:* is printed. The commands available are:

**access** file [file file .....]

The **access** command prints out a list of who may read, write, execute, and remove each file or directory listed on the command line. This list may be a single user's name, the name of a group, or a list of names. Although **access** is fairly smart about figuring out the most concise way to list the people who may do something to a file, occasionally it can't. If this happens, a line such as "there are 592 names in this list" will be printed, and you will be asked if you really want to see the list. If you do want to see the list, type 'y' (for 'yes'), otherwise, type 'n' (for 'no').

**cat** [args] file [file file .....]

Execute the **cat**(1) program on the named files. This is used when you want to see what a file contains. Some of the arguments **cat** accepts are **-n** to number the lines, and **-v** to print "invisible" characters.

**cd** directory-name

Change into the directory named *directory-name*. This command is just like the shell command of the same name.

**chmod** mode file [file file .....]

Execute the **chmod**(1) command on the named files. **Chmod** is used to change the permissions on a file. Modes are described in the manual for **chmod** and also in the help file for **access** (see the **help** command).

**exit**

Exit the **access** program.

**quit**

The same as **exit**. You may also type Control-D to exit.

## help

Display a help file listing all the commands available and their uses. This file is printed with the **more(1)** command. Typing a question mark ('?') also shows this file.

**info** [*args*] *file* [*file file .....*]

Run the **info(1)** command on the named files. **Info** prints out various pieces of information about a file or directory, such as its mode, owner, size, contents, creation time, etc. Some of the arguments to **info** include **-v** to print even more information, such as number of links, inode numbers, etc., and **-f** to skip trying to guess what's in the file.

**lock** [*file file file .....*]

Lock the named files. If no files are named, then the current directory is locked. **Lock** is a variant of the **chmod** command; it simply makes everything mode 0700 (readable, writable, and executable by the owner only).

**ls** [*args*] [*file file file .....*]

Execute the **ls(1)** command on the named files. If no files are named, the files in the current directory are listed. Some of the arguments to **ls** include **-l** to get a long listing, **-s** to show the size in kilobytes of each file, and **-a** to show files whose names begin with '.'.

**more** [*args*] *file* [*file file .....*]

Execute the **more(1)** command on the named files. **More** is similar to **cat**, except that it stops after every page of the file and waits for the user to press the space bar before going to the next page.

## pwd

Print the pathname of the current directory.

## sh

Execute a shell. The shell executed is normally taken from the environment variable **SHELL**, if this is not set, **/bin/sh** is used. To return to **access**, type Control-D (press the CTRL key and the D key at the same time).

## csh

The same as the **sh** command.

**unlock** [*file file file .....*]

Unlock the named files. If no files were named, unlock the current directory. This is the inverse of the **lock** command; it makes the named files mode 0755 (readable, writable, and executable by the owner, readable and executable by everyone else).

**Access**, even in interactive mode, understands the *metacharacters* used in the shell. That is, when naming files, the characters '\*', '?', and '[' have special meanings. These are described in the manual for the shell, either **sh(1)** or **csh(1)**. **Access** also understands the '~' character, which represents the home directory. For example, a '~' alone represents your home directory, but '~davy' represents the home directory for user "davy."

## How ACCESS Works

**Access** works by simulating (after a fashion) the **access(2)** system call for each user on the system. For each user, a structure is maintained containing the user's user id, login name, and a list of the groups for which he is a member. For each group, a structure is maintained containing the group's group id, its group name, and the number of users who are members of that group. The support program **mkaccessdb** is run nightly by **cron(8)** and is responsible for making the lists of users and groups and determining which groups each user belongs to. This information is then saved in two files which are loaded into memory each time the **access** program is executed.

The first thing **access** does to determine who may access a file is to figure out the full pathname to that file. This is done by obtaining the name of the current directory, and concatenating the name of the file to it. The concatenation routine is somewhat intelligent, and resolves things like "../" and "./" automatically. If a file's name begins with '/', the full pathname is already known, and the above procedure is not performed.

Now that the full path to the file is known, **access** begins "walking" the path and determining the protection mode of each component in the path using the **stat(2)** system call. At the outset, each user is "granted" permission to read, write, and execute any file. As the path

is traversed, the permissions for each user are modified according to the permission bits set for the current path component. These modifications are made following the same rules used by the operating system to determine access permissions. By logically *anding* the file's permission bits with those of the user, the new permission bits for the user can be determined. Note that this implies that while a user may *lose* permission to do something to a file, he can never *gain* permission to do something. The problem of which set (owner, group, world) of permission bits to use when determining a specific user's permissions is resolved by the following rules:

1. If the user whose permissions are currently being determined owns the file (or directory) being evaluated, his access permission is determined by the owner permissions on the file (directory). If he does not own the file, the group and world permissions will be checked.
2. If the user is a member of the group which owns the file, and he does not own the file, his access permission is determined by the group permissions on the file. If he is not a member of the group, the world permissions will be checked.
3. If the user does not own the file, and he is not a member of the group which owns the file, his access permissions will be determined by the world permissions on the file.

Note that under no circumstances will a user's access permission be determined by more than one set of permission bits on the file. In this way, he cannot regain access permission which has been denied by a previous set of permission bits.

Finally, after determining the access permissions for every user, **access** attempts to figure out a way to group together those users which have a certain access permission. For example, if all the users who have write permission on the file are members of the same group, then **access** can print "members of group x" instead of listing the users individually. Of course, if one of the users in the group does not have write permission on the file (e.g., if the owner permissions are read and execute and the group permissions are read, write, and execute and the owner is a member of the group), **access** would modify the message slightly to read "members of group x except y." Many other combinations are possible, and **access** is nearly always capable of finding one which fits.

In the few cases in which it cannot decide on a grouping (for example, when there are many sub-directories in the path to the file, each one owned by a different user and group, and file permissions which alternate between owner-only and "everybody"), **access** will offer to print the entire list of names. The user may decide whether he wishes to see this list.

The amount of processor time used by **access** can be divided into two parts: the time taken to read in the user and group files (a once per invocation occurrence), and the time taken to actually determine the access permissions for a file. For an "average" password file containing 107 users, **access** uses less than 0.3 seconds of processor (sys + user) time. For an extremely large password file containing 1,842 users, 0.2 seconds of user time and 1.1 seconds of system time are used. These times were recorded on a VAX-11/780 under 4.2BSD. This initial overhead (particularly with the large password file) prompted the creation of the interactive form of the command, enabling users to save time when checking a number of files by only reading the user and group files once.

After the user and group files have been read in, **access** runs very rapidly. To print out the access permissions for five files, each file five levels deep (i.e., /a/b/c/d/e), **access** requires 4.8 seconds of user time and 0.4 seconds of system time (in addition to the times used to read in the files) when running with the large password file; approximately 0.2 seconds of user and system time combined are needed when using the short password file. The times to print out the access permissions for the root directory ("/") are 0.4 seconds of user time and 0.1 seconds of system time for the large password file; less than 0.1 seconds of combined user and system time are needed when using the short password file.

### Special Cases

**Access** treats two conditions as special cases. First, **access** does not acknowledge the fact that user *root* has the capability to read, write, and execute *all* files, regardless of their owners or permissions. This treatment was decided upon in order to prevent users from becoming confused by some unknown account (root) who could always read, write, and execute their files.

Secondly, if **access** encounters a file owned by a user who is not in the password file, it assumes that the user does not exist (since it is

impossible to log in under that user id). Thus, if a mode 0700 (read, write, and execute for owner only) file is owned by a non-existent user, **access** will state that nobody can read, write, or execute the file.

### Conclusion

The **access** program has been well-received at the ECN, and is presently being taught to new students who are enrolled in the beginning computer course for the School of Electrical Engineering. By automating the process of determining who may access a file, the confusion for the user has been removed. Although this is arguably wrong, since the user will probably never learn how the permissions actually work, it seems to be the most desirable solution. This is primarily because most of our student users use the machine for coursework only, and are not interested in how things work, only in getting their work done.

This document and the software it describes are hereby placed in the public domain and may be used by anyone for any purpose provided that they are not used or sold for profit and that this notice and the names of the original authors appear with all copies.

# A High-performance Model for 2-D Alphanumeric Display Generation

Paul Bame  
Software Development Engineer  
Hewlett-Packard  
P.O. Box 617  
Colorado Springs, CO 80901

## ABSTRACT

Typical two-dimensional display schemes in UNIX are often awkward extensions of the stream-oriented tty model. That is, two-dimensional directives are encoded into a stream representation for directing the target display. Much higher performance can be realized on integrated workstations and intelligent terminals (aka 'BLIT') by eliminating the overhead of typical escape sequence parsers and line discipline. HPFastAlpha attempts to provide this high performance in a simple, straightforward manner. By carefully defining a small, generic set of display manipulation functions, HPFastAlpha is a fast and portable interface to 2-D alphanumeric displays.

## The Past

Traditionally, two-dimensional displays in UNIX have been constructed using the BSD curses library. Among those applications are editors, menu systems, and rogue. Curses uses the /etc/termcap database and library routines to control the target display via escape sequences. These escape sequences do not contain high-level directives, such as arbitrary rectangle scrolling.

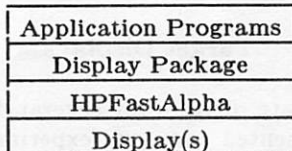


Figure 1: Structure of Programs Using 2-D Displays

On an integrated workstation, the escape-sequence parsing and line discipline reduce the performance - sometimes drastically.

Although curses provides a method for constructing two-dimensional displays, it is not sufficient for some applications. Curses is neither fast enough nor flexible enough for many of HP's instrumentation needs. To solve this problem, we defined and implemented a "display package" with some features similar to curses, without the curses input functions, and with a much more flexible windowing facility. Our "display package" is a high-level, device-independent display

interface [see figure 1]. In fact, even the "display package" itself is device-independent because it is built upon the device-independent HPFastAlpha interface.

## HP-Fast-Alpha Model

### Investigation

The "display package" required high-performance and device independence. An application program (i.e., an editor) should be capable of programming for workstation displays and terminals without affecting source code. In order to modularize the "display package" software and minimize the impact of new display drivers, a low-level display interface was needed.

In the search for a suitable low-level display interface, we examined the capabilities of existing display devices. After several iterations, it became apparent that all the functions commonly performed by terminals (i.e., delete to end of line, roll, insert line, insert character, position cursor, etc) could be described with four basic functions.

### Description

HPFastAlpha is an output-only interface with a generic set of display capabilities. This small set of functions can easily be implemented for any random-access target display capable of

displaying alphanumerics. All device-dependent functions are hidden in the HPFastAlpha driver for the particular devices.

### Four Basic Functions

Four functions form the basic set required to generate two-dimensional displays. More detailed function descriptions are given later in this document.

`facursor(col, line, cflag)`

This function positions the cursor and/or turns it on or off. Note that this is the only function that modifies the cursor state.

`fawrite(col, line, data, length)`

The indicated data is written at the display position specified.

`faroll(how, howfar, rectangle)`

An area on the display designated by **rectangle** is rolled within that rectangle by **how** many units in the direction denoted by **howfar**.

`farectwrite(c, enhancement, rect)`

This is a mnemonic for "rectangle write". The rectangle denoted by **rect** is filled in with the character, **c**, and **enhancement** specified.

### HPFastAlpha is Enough

This set of simple functions is enough to accomplish all common display manipulations. It is instructive to examine "insert line" as an example. The "insert line" function on an HP terminal rolls the area from the line containing the cursor down one line. The cursor is moved to the beginning of the now blank line. Here is a scrap of pseudo-code implementing this function.

```
/* cursor is on line 'cursorline' */
rectangle_to_roll =
    {0, cursorline} [MAXCOLUMN, MAXLINE]
/* open the new blank line */
faroll(fd, FAROLLDOWN, 1, &rectangle_to_roll);
/* and position cursor */
facursor(fd, 0, cursorline, TRUE);
```

Another simple function is clearing to the end of a line. In this operation, the cursor position and all characters to the end of the line containing the cursor are blanked.

```
/* cursor is at [cursorcolumn, cursorline] */
rectangle = [cursorcolumn, cursorline]
            [MAXCOLUMN, cursorline]
/* blank to end of line */
farectwrite(fd, ' ', FAOFF, &rectangle);
```

### Advanced Uses

#### User Interfaces

For applications requiring a complete user interface, HPFastAlpha must be augmented with an input capability. In UNIX, input may be provided with the existing tty driver. Other operating systems require other input routines.

#### Color

Generic support for fixed-width fonts and color is part of HPFastAlpha, but these functions may be safely ignored. The color model assumes some type of foreground and background color capability. The `fasetcolor()` function changes these colors. A suitable default color is active at initialization, so colors may be disregarded altogether.

#### Fonts

The font model requires the programmer to choose fonts with identical cell sizes. Three functions `fontload()`, `fontactivate()`, and `fontremove()`, determine the characteristics of the font used. A suitable default font is active at initialization time, so, for this application, fonts may be disregarded altogether.

#### Target Displays

To date, drivers for several displays have been implemented on an experimental basis. These drivers are implemented as libraries. An application may link to either the specific device-

Device-dependent Driver
----------------------------

Figure 2: Device-dependent HPFastAlpha Library

dependent driver needed [see figure 2]. or to a master library that offers a selection of several

FastAlpha Driver Switch		
Driver-1	Driver-2	Driver-N

Figure 3: Master HPFastAlpha Library

drivers [see figure 3].

### Alpha-plane Workstation Driver

This driver is written for the HP-9000 series 200 Model 236U workstation that has an alpha-plane display architecture, similar to most normal terminals. Because of this architecture, it is very fast. The driver software occupies less than 3k bytes of 68000 text without color or multiple font support.

### Generic Terminal Driver

One of the current experimental drivers uses the TERMCAP database to drive almost any display. This is similar to curses in that optimization to minimize serial stream length is performed.

### Bitmap Driver

An experimental driver for a high-resolution bitmapped device (HP-9000 series 200 Model 220U with an experimental display) shows promise for higher performance there also. Since the HPFastAlpha model is output only, no duplicate ASCII screen buffer is needed. Applications requiring ASCII buffers (such as terminal emulators) may add those buffers outside of HPFastAlpha.

In the future, integration of HPFastAlpha capabilities with a windowing system will provide a simple, convenient, backwards-compatible method for talking to windows with highest possible performance.

### The Future

#### Curses Rewrite - Fast Rogue

Curses can be rewritten in terms of HPFastAlpha. If this were done, applications currently using curses would run very fast on suitable displays. Imagine rogue or EMACS at 10-100k bytes/second effective screen speed!

#### Other Target Displays

Personal computers with local computation power are being connected to other computers and used as terminals. Given the amount of local intelligence available, the complete HPFastAlpha functionality could be implemented in the personal computer for higher performance than possible in a standard terminal. Other targets for HPFastAlpha are terminals with local windowing

capabilities (WE5620, aka BLIT).

HPFastAlpha is not UNIX dependent and can be implemented on many hosts. This provides a convenient method for increasing the portability of software to and from other operating systems (e.g., MS-DOS).

### Quest for Standardization and Portability

We believe strongly that some standards are necessary to increase portability among the widely diverging window systems while remaining compatible with old alphanumeric terminals. If the HPFastAlpha interface is adopted as an industry standard, implementations for current and future display devices and reimplementations of current software systems (e.g., the Maryland Window System curses) would increase software portability markedly.

### Nonalphanumeric Applications

On devices capable of displaying user-defined fonts, HPFastAlpha provides a convenient method for high-performance display manipulation. The HPFastAlpha standard interface could replace the nonstandard, low-level bitmap driver. Possible applications include chess programs and schematic capture systems.

### Status Of HP-Fast-Alpha

HPFastAlpha is being used experimentally within Hewlett-Packard. It has been approved by several divisions for internal use.

### Additional Information

Complete working manual pages containing more detailed function descriptions may be obtained from the author. To contact the author use:

UUCP-mail: {ihnp4!hpfc!hplabs!}hp-bsd!paul  
CSNET: hp-bsd!paul@HP-LABS  
ARPA: hp-bsd!paul%HP-LABS@CSNET-RELAY  
TELEPHONE: (303) 590-3103 (Mountain Time)  
US-Mail: Paul Bame, P.O. Box 617, Colorado Springs, CO, 80918.

### Function Descriptions

```
int fainit (fd, driver);  
    int fd;  
    char *driver;  
  
faterminate (fd);  
    int fd;
```

These functions start and terminate use of the HPFastAlpha functions. **Fd** is assumed to be connected to an open device (i.e., stdout) and **driver** is a text string used for device-dependent driver selection. **Driver** is commonly assigned the value of the **TERM** environment variable. **Fainit** is constructed such that **fainit(fileno(stdout), getenv(TERM))** prepares HPFastAlpha to operate upon the device seen by the user as their standard output device (which could be a window). This has implications in some architectures, where, for example, stdout references a pty device, different from the window device that actually appears as the standard output.

```
facursor (fd, column, line, cflag);
    int fd;
    int column, line;
    int cflag;
```

**Facursor** positions the cursor and/or turns it on and off.

```
fagetinfo (fd, fainfoptr)
    int fd;
    struct fainfo *fainfoptr;
```

```
fasetinfo (fd, fainfoptr)
    int fd;
    struct fainfo *fainfoptr;
```

This determines device characteristics, such as size, color capability, and enhancements available. Some fields may also be written to change device behavior.

```
fawrite (fd, column, line, charbuf, ebuf, nchars)
    int fd;
    int column, line;
    char *charbuf;
    ENH *ebuf;
    int nchars;
```

**Fawrite** is the primary HPFastAlpha routine. It is used to write characters (with optional enhancements) at arbitrary places on a display. A character count is used (instead of the normal C-language null-character termination) to allow writing of nonprinting characters.

```
faroll (fd, how, howfar, rp);
    int fd;
    int how, howfar;
    struct fa_rectangle *rp;
```

The rectangular area on the display described by **rp** is rolled as indicated.

```
farectwrite (fd, character, enhancement, rp);
    int fd;
    char character;
    ENH enhancement;
    struct fa_rectangle *rp;
```

The character described by **character** and the enhancement described by **enhancement** are written into the area described by **rp**. This is commonly used to clear a display (e.g., **character** is a blank).

## Monitoring System and Process Performance

William J. Meyers

SCI Systems, Inc.  
P.O. Box 12557  
Research Triangle Park  
North Carolina 27709  
(919) 549-8334  
decvax!mcnc!rti-sel!scirtp!bill

### ABSTRACT

Recent versions of UNIX\* maintain system performance data in kernel memory, and per-process cpu time and I/O statistics in the user block. We wrote simple utilities to display this information as dynamically updated bar graphs.

Process Monitor (pm) displays current cpu and I/O activity for each process on the system. When updating the display every 10 seconds, it uses about 6% of cpu time and 8% of I/O capacity on an 80186 UNIX system. Besides providing something interesting for people to watch, pm has identified a couple of "burn cpu while waiting for signal" bugs, and proved helpful in debugging software that uses interacting processes.

System Monitor (sm) displays information from the sysinfo structure. This varies from one system to another, but generally covers at least cpu usage, system calls, scheduler activity, filename resolution, and character, block, and terminal I/O. When updating the display every 5 seconds, sm uses about 2% of cpu time and 1% of I/O capacity on an 80186 UNIX system. It confirms or denies that our system is swapping, and is expected to be useful while running benchmarks.

Both utilities run as privileged applications, like ps, and use the termcap data base for terminal handling. In order to port pm to another system one need only find the process table and user blocks (say by reading ps.c), and determine the new clock rate and I/O capacity (for which pm includes calibration aids). Because sysinfo varies so much, sm is primarily table driven, and includes a learning mechanism to calibrate itself. In order to port sm to another system one need only find sysinfo, decide what to display, and alter the tables accordingly.

---

\*UNIX is a trademark of AT&T Bell Laboratories, Inc.

## Monitoring System and Process Performance

William J. Meyers

SCI Systems, Inc.  
P.O. Box 12557  
Research Triangle Park  
North Carolina 27709  
(919) 549-8334  
decvax!mcnc!rti-sel!scirtp!bill

### 1. Introduction

There are many occasions to wonder how a system's resources are being consumed. Accurate and timely information on the subject can help identify bugs in both hardware and software, and tune both systems and applications. For example, a constantly active login process might indicate a noisy or miswired tty line. A process known to be waiting for a signal, but still consuming cpu time, would probably represent a software bug. Just watching things work helps build insight.

Recent UNIX systems maintain information on both per process and system resource consumption, in the user blocks and a kernel memory structure respectively. We wrote two utilities to display this information graphically. Process Monitor (pm) displays cpu and I/O activity for each active process, by sampling user blocks and differencing the information in successive samples. System Monitor (sm) does the same thing for the system performance data in the "sysinfo" structure. Both utilities sample every few

seconds, and display their results as bar graphs updated in place on a terminal screen.

In the following we describe the function and general design of pm and sm. The interesting points are design of the pm display, the two parameter bar graph format, reading accurate process data without help from the system, and table driving the sm display. We also discuss tradeoffs between portability, accuracy, and performance.

### 2. Design of the Displays

An obvious prototype display for pm is the one given by ps, which simply writes out process information in the order in which the processes appear in the proc table. However, this format is awkward for a repeating display because processes jump up and down as process table slots ahead of them empty and fill up again. We wanted a display format that would

[a] tend to repeat the display for each process in the same place, and

[b] put new processes where they would be noticed easily.

The first characteristic also makes it reasonable to update the display selectively, in order to reduce pm's own I/O activity.

We decided to display processes in order of seniority, with ties broken by pid number. This puts older processes at the top of the display, and newer processes at the bottom. As older processes terminate and leave the display, newer processes gradually move upwards. Most of the display remains quite stable, since short lived processes stay near the bottom.

In addition to giving a more readable display, ordering processes by seniority might be more appropriate on a distributed system. The idea is not without disadvantages, however, since it assumes that every process on the system has a reasonable start time. Processes 0 and 1 are occasionally neglected in this respect, in which case pm has to know about it and compensate.

For readability each process was also restricted to one line of information on the display. This eventually boiled down to

pid number,  
effective user name,  
command file name, and  
bar graph of cpu & I/O use.

The bar graph format represents both cpu and I/O use on a single line by superimposing a cpu bar graph and an I/O bar graph, using the following hack. Represent each quantum of cpu by a '-', each quantum of I/O by a ':', and

their overlap (if any) by a '+'. Then differently balanced jobs display as follows.

----- (all cpu)  
++++----- (mostly cpu)  
+++++++ (evenly balanced)  
++++::: (mostly I/O)  
::: (all I/O)

Thus the thin line through the middle represents cpu time, and the fat line around it represents I/O activity. Of course the lengths of these lines are scaled relative to the maximum cpu and I/O capacity of the system, within the line length available on the user's terminal.

Terminal standout or underlining features might have given a more attractive display on some terminals, but would have been more trouble and less portable.

The ability to graph two quantities on one line proved essential in sm, which describes the 32 sysinfo fields (for our system) on a 24 line terminal. We put related pairs on the same line, and used the extra room to organize the display into broadly related resource groups: cpu and scheduler activity, swapping, filename resolution, and I/O.

### 3. Description of the Programs

Both pm and sm periodically sample resources of interest, and display differences between the current and previous samples. They display to stdout, complain (if necessary) to stderr, act appropriately when forked or piped, and normally run until interrupted or killed. On our system they run with the same

privileges as ps, since both read information from kernel memory.

In order to update the bar graphs in place, some form of "home" terminal function is essential, and "clear screen" and "erase to eol" are convenient. Both pm and sm use the term lib functions to get corresponding terminal control sequences (and screen dimensions) from the termcap data base. About 10% of their code involves terminal independence. They check for the same terminal capabilities as vi visual mode (i.e., "move", "home" or "up"), and refuse to run if none of these are available. Of course one could do something reasonable on dumb terminals, hardcopy, storage tubes, etc., but we didn't consider it worth the hassle.

### 3.1. Process Monitor

Within the general framework described above, pm carries out the following special functions.

- [1] Sample per process data.
- [2] Sort into system snapshot.
- [3] Measure process activity.
- [4] Display.

The details are discussed below.

#### 3.1.1. Sample Per Process Data

The most basic thing that pm has to do is to read identity and accounting information for one process. This takes some knowledge of UNIX internals, specifically the proc table and user blocks. The easiest way for non experts to locate these structures is to read the source code for ps or fuser.

Early UNIX systems only kept track of per process system time and user time. Bell Systems III and V also keep per process I/O information, but Berkeley 4.2 doesn't. If your system doesn't keep per process I/O information, pm can't display it.

To read process accounting information, we wrote a procedure that takes a proc table slot number as argument, reads the corresponding proctable entry, and returns 1 or 0 according to whether it does or doesn't represent an active process. If there is an active process the procedure also reads the corresponding user block, copies process identity and accounting information into a structure defined for that purpose, and time stamps it. The time stamp allows for irregular delays due to system load or running nice, and makes the program logic independent of changes in the sampling interval.

The standard time function has a resolution of one second. When sampling every 10 seconds, therefore, the time stamp could have up to 10% quantization error. However, the final result of pm's calculations is a bar graph, typically displayed in a field of about 50 characters. Thus an error of less than 2% would be imperceptible, and 10% wouldn't look like much. Time to the nearest tick would be an improvement, but more precision than that would be lost in the display.

Since pm reads the proc table and user blocks without any particular cooperation from the operating system, many uncertainties have to be dealt with. Per process data changes on the fly, for example, although one can

(apparently) take a consistent snapshot. A more serious problem is that the user block moves around when a process swaps, which it may while pm attempts to read its user block. Therefore anything that casts doubt on the whereabouts of process data -- a fork in progress, for example, or a change in proc table flags -- is considered grounds for trying the read again. Before we used these precautions about 1 in 10,000 reads missed the user block, and propagated trash into the display. Now there are too few misreads to analyze, although it is easy to construct scenarios in which they occur.

### 3.1.2. Sort into System Snapshot

The next step is to combine the identity and accounting information for all active processes into a snapshot of current system activity. Another procedure does this by calling the per process routine for each slot number in the proc table, and sorting the results into a larger structure. If the sort encounters aberrant data -- a process "older" than process 0, for example -- it double checks the call for information about that process. We use insertion sort into a linear linked list, since our system seldom has more than 20 active processes. Bigger systems might justify a better algorithm.

### 3.1.3. Measure Process Activity

The last step before actual display is to search the current and previous system snapshots for corresponding process records, and difference their time stamped accounting information. Separate "cpu" and "io" functions are used to calculate the cpu and I/O rates from each pair of process

records. The results for each process are stored, along with process identity, in a structure passed to the display system.

Having the system snapshots already sorted by process age facilitates the search for corresponding process records, but makes it sensitive to bad start times caused by misreading the user block. We've had only one confirmed report of this problem in 3 months, but it might be interesting to experiment with more robust ways of identifying the process records.

On our UNIX system the user blocks have 3 fields of interest to the cpu function, and 4 fields of interest to the io function. In addition to boiling down this information, the functions sanity check their data, correct some errors, scale the results, and log anomalies to a /tmp file. For example, they don't believe in negative numbers. On the other hand, positive numbers larger than the presumed system cpu or I/O capacity could be either trash or good data, and are logged as questionable. The log file can therefore be used to recalibrate pm for new maximum cpu and I/O usage rates, when porting it to a new system. This is now the principal use of the log file, since pm seldom misses the user block any more.

The cpu and io functions scale their results into the range 0..32767, making it unnecessary for other routines to know the maximum system cpu and I/O rates. (Similarly, the bar graph display routine scales its parameters from this range back down to the line length of the user's terminal, making it unnecessary for other routines to know that.) Although 32767 may be

too small to count characters per second exactly on a big computer, the 15 bits of precision is more than adequate to drive a 50-100 character bar graph display.

#### 3.1.4. Display

The display system is passed an array that contains each process's identity, cpu rate, and I/O rate, in the order in which they should be displayed. It simply walks down the array displaying what's there. Both current and previous copies of the display information are kept, so the routine can rewrite only information that has changed since the previous display.

Since each terminal has a fixed number of lines, usually fewer than the number of process table slots, it's easily possible for pm's display to exceed the screen size. Rather than scroll (and trash the smart update), pm simply truncates its display. This created a need to scroll the display window, which we met by using non-blocking reads from stdin. The user may type in "u" or "d" characters asynchronously, to move the display up or down in the terminal window on the next update. This required altering stdin's I/O characteristics (if a tty), which in turn required catching signals to set stdin right again -- recall that pm normally runs until interrupted or killed. About 10% of pm's code involves non-blocking I/O. Kill -9 leaves things messed up, of course, and shouldn't be used except in emergencies. Parent shells left with non-blocking I/O on stdin will immediately read 0 characters and terminate on the presumed eof, thus logging off.

### 3.2. System Monitor

In general outline, sm is very similar to pm. They use the same sampling, differencing, and display techniques, but sm reads the sysinfo structure instead of process data. It carries out the following special functions.

- [1] Sample sysinfo.
- [2] Measure system activity.
- [3] Display.

The details are described below.

#### 3.2.1. Sample Sysinfo

The performance information in sysinfo is sampled by simply reading a copy of the sysinfo structure from kernel memory, and time stamping it. Of course this time stamp is subject to the quantization errors discussed in Section 3.1.1 above.

#### 3.2.2. Measure System Activity

System activity is measured by differencing the corresponding fields of current and previous sysinfo snapshots. On our system there are 32 fields covering various aspects of cpu time, scheduler activity, system calls, swapping, file name resolution, block I/O, character I/O, terminal I/O, and interrupts. Each of the fields is measured in one of the following units per second.

ticks (60-ths of a second)

process switches

system calls

segments swapped in/out

directory blocks read  
file blocks read/written  
characters read/written  
terminal characters r/w  
interrupts

For each of these units sm keeps a maximum value (per second), representing the system capacity for the corresponding resource. These maximum values are updated whenever sm encounters larger ones. In effect sm calibrates itself, by "learning" the system capacity for each resource. This makes sm more easily portable.

Most of the maximum capacity values are used for several resource measurements apiece. For example, swaps in and swaps out are both scaled relative to the same maximum number of segments swapped per second. This makes it apparent from the display that the system normally swaps in more than it swaps out. Similarly, there tends to be more file input than output, but more tty output than input. It seems obvious on reflection that this is how a system should behave: the point is that sm's scaling mechanism portrays it accurately.

The correspondence between the various resource measurements and their maximum capacity values is handled by about a page of inline code with 32 calls to a scaling function. Each call has the form

`scale(&max, cur, pre, dt)`

where "cur" and "pre" are the current and previous measurements of a particular resource, "dt" is the difference between their time stamps, and "max" is the maximum

capacity value for that resource. Scale calculates  $(cur-pre)/dt$ , normalizes it from the range 0..max into the range 0..32767, and updates max as a side effect if  $(cur-pre)/dt$  was greater. Although this mechanism lacks elegance, it will accomodate any reasonable variation of sysinfo from one system to another.

### 3.2.3. Display

The results of scale are placed in an array, and passed to the display system. There is a parallel array of display tags, which describe the corresponding system resources. The display routine simply walks down both arrays, displaying each resource's tag and current value. Both current and previous copies of the value array are kept, so the display routine can update only information that has changed since the previous display.

## 4. Performance

We used pm to measure both its own performance and that of sm on our 80186 UNIX system. When sampling every 5 seconds, sm uses about 2% of the cpu time and 1% of I/O capacity, which is negligible. When sampling every 10 seconds, pm uses about 6% of cpu time and 8% of I/O capacity, about 3 times the cpu and 8 times the I/O of sm. Evidently reading all those user blocks takes its toll. Since pm needs very little of the information in the user block -- less than 10%, on our system -- one might try reading only that part. This would make pm more complex and possibly less portable, but reduce I/O by a factor of 10, and cpu by the time it took to move that data. Then pm would use perhaps twice the resources of sm, which seems more reasonable.

## 5. Conclusion

Both pm and sm are fun to watch and informative. The graphic displays are easy to grasp, and the information is relevant to users' interests. The same techniques could be used to monitor many other system phenomena. One that comes to mind immediately is network traffic. Another example that might be useful is the percentage of available inodes in a UNIX file system. The only limits are thinking of what to measure, and

arranging for the controlling processes to leave relevant data in an accessible place.

## 6. Acknowledgements

Video displays of system activity are nothing new, of course. I first saw a continuous bar graph display of per process resource use done by Fred Harms, at Data General Corporation, about 1979. Thanks are also due to Bob Eifrig, of Interactive Systems Corporation, for much useful UNIX advice.

```
0  root  kernel
1  root  init
21 root  cron
25 root  qdaemon
36 root  qftp
45 bill  sh
46 root  sh
201 tom  sh
463 root  getty
464 bill  grep -----
465 bill  pm  ++-
466 bill  tail +
```

Example of a pm display: grep is getting most of the cpu time.

```
0  root  kernel
1  root  init
21 root  cron
25 root  qdaemon
36 root  qftp
45 bill  sh
46 root  getty
48 root  sh
62 root  getty -
77 bill  dd  ++++++++.....
78 bill  pm  ++-
79 bill  tail +
```

Example of a pm display: dd is using most of the I/O capacity.

```

cpu/scheduler idle
  kernel/user cpu ++++++:::
    system calls -----
    process switches ---
process quit/preempt +:

```

```

    waiting for swap
  seg/block swaps in ++++++-----
  seg/block swaps out ++++++-----

  pathname searches -
dir block reads/hits +

  waiting for io/dma
    process byte r/w +---
    logical block r/w ++++++---
    physical block r/w +-
    hard disk block r/w ++++++-----

  raw/can tty input
    tty output -

rcv/xmt interrupts :
  modem interrupts

```

Example of an sm display: this system is swapping vigorously.

```

cpu/scheduler idle +-----
  kernel/user cpu +-----
    system calls ---
    process switches -
process quit/preempt :

    waiting for swap
  seg/block swaps in
  seg/block swaps out

  pathname searches ---
dir block reads/hits ++++++-----

  waiting for io/dma -----
    process byte r/w +
    logical block r/w +-----
    physical block r/w +---
    hard disk block r/w +---

  raw/can tty input
    tty output ---

rcv/xmt interrupts :::
  modem interrupts

```

Example of an sm display: scanning through a large directory.

## Interpreting UNIX benchmarks

John Saxer  
CIE Systems

Benchmark data confronts the buyer and users of computers with increasing regularity every day, particularly in the UNIX computer systems market. Every vendor, both hardware and software, has a benchmark of one kind or another to "prove" that his machine or implementation of UNIX is better or faster than the next one. How are you to evaluate this confusing mass of data? Are there any standard benchmarks that can be applied? Are the numbers presented accurate, and what exactly do they mean? Does the benchmark actually measure what it is purported to measure? The following discussion can help you answer these questions for benchmarks running under UNIX and UNIX-like operating systems on many different types of hardware.

UNIX invites benchmarking because the basic method of measuring performance, elapsed time, is relatively easy to obtain. The UNIX standard 'time' command allows even novice users to measure the performance of anything they desire, which is often what occurs. Since the 'time' command is basic to our measuring of benchmarks, it is important to understand how this command works and what it tells you, and how accurate are the values that it will report.

### The TIME command

The UNIX 'time' command depends upon two sets of values maintained by the operating system. The first value is the time of day clock, which is incremented once every second. The 'time' command reads this value before starting the task to be timed, and again after the task completes. The difference of these two values is reported as the "real", or elapsed, time that the task required.

An error of up to 2 seconds is possible, simply by reading the clock value each time immediately prior (one clock tick) to it being incremented. On systems running 4BSD UNIX, this is accounted for by returning a value for micro-seconds. This value is not completely accurate, and should be rounded up for compatibility sake.

In addition to this error, there can be a considerable delay between the time of the first reading and the time that the task actually starts, and another delay between the time the task ends and the final reading. This is due to the fact that the 'time' command must fork and execute the new task, and then wait for it to finish. Waiting may also result in the 'time' command being swapped out, since waiting tasks are the primary candidates for swapping.

The error resulting from these inaccuracies may be minimized in two ways. The first is to run every benchmark several times, and take the average of the results. This should effectively remove the error due to accessing the clock at an unknown fraction of a second.

The second method is to determine the startup and processing overhead of the 'time' command itself. This can be done by running the benchmark given as PROGRAM #1. The program does no actual work, simply starting and ending. On most processors and operating systems, the elapsed time value should be close to zero. Any other value must be considered when examining benchmark data.

#### PROGRAM #1

```
main() { }

% cc -o pgml pgml.c
% time pgml
```

The other 2 values reported by the 'time' command are the amount of 'system' and 'user' time used by the task being timed. These values are maintained by the operating system for each child task of a process. Each time the clock interrupts the processor, the operating system increments the value associated with the state where the processor was currently executing.

The accuracy of these values depends upon the frequency of the real-time clock. It is therefore important to know the clock frequency of machines being benchmarked, and to determine the amount of time required to service a system call. The DEC PDP-11 and VAX computers use a 60Hz clock, while many micro-computer systems use clocks with rates that may be as low as 10Hz or 16Hz.

#### System Call Overhead

PROGRAM #2 gives us a method of determining the amount of time required to make a system call. Note that the design of the program does not attempt to determine the amount of time required to service system calls, since this will vary considerably depending on the system call. The program is instead designed to give us the average overhead required to make the system call.

## PROGRAM #2

```
#ifndef WORK
#   define WORK getpid()
#endif

#ifndef CONSTANT
#   define CONSTANT 10000
#endif

main()
{
    register int i;

    for( i = 0; i < CONSTANT; i++ )
        WORK;
}

% cc -o work pgm2.c
% cc -o nowork -DWORK pgm2.c
% time nowork ; time work
```

Since we must make a system call to perform this test, we choose one that requires little or no actual work for the operating system. In UNIX, the process id of the currently executing task is available from the user vector (a structure maintained by the system for every task). Therefore, the 'getpid' system call is a reasonable choice for this test.

When compiled into the program "nowork", we have a benchmark that measures the overhead of running the loop. This overhead must then be subtracted from the time of the program "work", to determine the actual system call overhead. As might be expected, increasing the loop constant will improve the accuracy of the results.

To calculate the system call overhead, subtract the time for the "nowork" program from the "work" program, and divide by the loop constant. We use the "real" time because of the correction provided by "nowork".

Why should we measure system call overhead? Since every system service required by a process is requested via a system call, the amount of overhead directly affects the execution speed of every program. As we will see with later benchmarks, the affect of system call overhead can be the determining factor in the overall speed of certain functions.

### 'C' Compiler Benchmarks

One of the most common benchmark results that are reported is not the system call performance of the above test, but the 'C' compiler performance in generating the test program. To the world of program developers who have traditionally embraced UNIX, this

may be an important factor. But the new breed of UNIX users are business or word-processor users, for whom compiler efficiency is of little practical use. It may be claimed that the compiler provides a measure of the systems efficiency because of the range of services that it requires, but is this true? A close examination of the functioning of the average compiler should give us a reasonable idea of the validity of these claims.

The compiler under UNIX is typically comprised of 3 or 4 separate programs, each called by a driver program called 'cc'. The other parts of the compiler are the preprocessor, which handles include files and defined constants and macros; the parser, which performs syntax checking and constructs the parse tree; the code generator, which generates the assembler code; an optional code optimizer; the assembler, which produces the object code; and the loader, which takes the object code produced and loads it together with library subroutine modules to produce an executable program.

As you can see, the main driver process does little other than construct temporary file names and execute the other passes. (In some older versions of the UNIX compiler, the preprocessor phase was included in the main driver process.) Let us consider the amount of work required by the other passes.

The preprocessor expands definitions and macros, processes include files and strips comments. In the benchmark program, we have only 2 defines, no macros, no include files and no comments. The work required would appear to be the simple copying of the input file to a temporary file.

The next step, the parser, parses 'C' language constructs into an intermediate language used by the code generator. With no complex data structures or language constructs, the parser has very little to do for this example.

The code generator translates the intermediate language of the parser into assembly language statements. The example program produces between 20 and 40 lines of assembler output.

The assembler translates the assembly code into object code. Whether 1 pass or 2 pass, the assembler reads the single input file and produces a single output file which varies in size from 200 bytes to 300 bytes.

The amount of I/O for these 4 programs is therefore fairly small, with temporary files almost certainly less than the 512 bytes that will fit any UNIX version's block size. As a test of I/O, these passes of the compiler fail miserably.

The loader is the final pass of the compiler. The loader takes the object file, standard header object file, and a standard library and resolves the external references, producing a relocatable object program capable of being executed. The library ranges in size from 50K to 65K bytes.

The original UNIX loader scanned the library from top to bottom, resolving external references as it went. UNIX versions from 4BSD and SYSTEM V support library formats with symbol directories at the beginning, limiting the amount of I/O to whatever is required to actually read in the required modules. Our benchmark program requires from 8 to 20 modules from the library, many of the modules (those for system call interfaces, particularly), less than 100 bytes long. Therefore, as a test of I/O, even the loader is no longer a useful process.

The major value of compiler timings is in the process creation and process termination times for the various pieces. As we have seen, this value can be better determined in another way. To test I/O throughput, we need another type of test.

## I/O

The main area of UNIX that is commonly benchmarked is I/O, since most programs that do useful work normally perform some sort of I/O. In UNIX, there are three types of I/O that should be studied. These are file I/O, character (tty) I/O, and pipes. File I/O is generally broken into serial and random access I/O, and block and non-block I/O.

Of these types, character input is the most difficult for the casual user to benchmark, since it requires a second computer to generate a known input stream. Character output is easier to benchmark, but would require several terminal devices at the same time. Since this is beyond the ability of the novice user, we will omit this type of I/O from our discussion.

## FILE I/O

Files under UNIX are defined to be a stream of bytes, with no order or ordering provided or enforced by the operating system. The knowledgeable UNIX user knows, however, that the stream is blocked by the operating system for practical purposes of writing to disks. The size of the blocks varies from system to system, and from version to version. In older versions, a block was 512 bytes. Newer versions use blocks of 1024 bytes, and may go as high as 4096 bytes.

In considering block I/O benchmarks, it is important to use the block size of the operating system being studied, since the design of the benchmark is to determine the throughput of blocks, and not necessarily the number of bytes. It is to be expected that a system using 1024 byte blocks will be able to write more bytes per operation than one using 512 byte blocks, but the amount of work involved in performing the write may not be as proportional. The block size of the operating system is usually available to programmers as the defined constant 'BUFSIZ' in the file `"/usr/include/stdio.h"`.

In addition, it is important to insure that the blocks being written actually contain data. UNIX supports the idea of sparse

files, which means that blocks within a file that consist of only zeros may not actually be allocated within the filesystem. This may or may not be implemented on all current ports of UNIX, and may in fact be implemented differently on different versions. To guard against generating sparse files, all I/O benchmarks should force at least 1 byte of non-zero data into the buffers being used.

PROGRAMS #3 and #4 are an example of a common serial I/O benchmark. The first consideration in evaluating these benchmarks is whether they have any applicability to the real world. Under UNIX, the answer should be a resounding yes. Almost every filter program in UNIX processes its data serially, and they are usually written to use block I/O. The UNIX philosophy of filters insures that much of the work in many installations will be of this type.

What can be learned from these benchmarks? The write benchmark (PROGRAM #3) gives us a measure of the amount of work involved in file system free list manipulation, in inode (or its equivalent) manipulation, and in disk throughput. The read benchmark (PROGRAM #4) gives us a measure of inode access, and disk throughput, and operating system read-ahead performance.

Outside of disk throughput, which is dependent upon the amount of money that is spent on disk hardware, the most important items to measure are free list manipulation and read-ahead. Since programs under UNIX tend to create, read and write temporary files, freelist manipulation is a vital consideration.

We can isolate the freelist manipulation overhead using PROGRAM #3 compiled to give the 2 programs 'create' and 'write'. The difference between the running times of these 2 programs should be a direct reflection of the freelist manipulation overhead, which includes searching, reading and writing of the freelist, and allocating the blocks to the file.

It is important to remove the test data file before running the create benchmark, to insure that any filesystem manipulation required to delete the previous contents of the file (which would be a consequence of the 'creat' system call), are not added to the work being benchmarked.

### PROGRAM #3

```
#include <stdio.h>
#ifdef BSIZE
#   define BSIZE BUFSIZ
#endif

main()
{
    register int fd, i;
    char bur[BSIZE];

    #ifdef CREATE
        *bur = '1';
        if( (fd = creat("test",0644)) == -1 )
    #else
        *bur = '2';
        if( (fd = open("test",2)) == -1 || lseek(fd,0L,0) == -1 )
    #endif
        exit(perror("test"));
    for( i = 0; i < 500; i++ )
        write(fd,bur,BSIZE);
    close(fd);
}

% cc -DCREATE -o create pgm3.c
% cc -DWRITE -o write pgm3.c
% rm -f test ; time create ; time write
```

### PROGRAM #4

```
#include <stdio.h>
#ifdef BSIZE
#   define BSIZE BUFSIZ
#endif

main()
{
    register int fd, i;
    char bur[BSIZE];

    if( (fd = open("test",0)) == -1 )
        exit(perror("test"));
    for( i = 0; i < 500; i++ )
        read(fd,bur,BSIZE);
    close(fd);
}

% cc -o read pgm4.c
% create ; time read
```

Disk throughput is calculated using the results of the write program, subtracting the base time of program #1, and dividing by the number of blocks written. The effects of read-ahead, if any,

can be calculated using program #4. Subtract the base time from the time for program #4, and divide by the number of blocks read. This will give a throughput value for reads. The difference between the write program throughput and the read program throughput should be due to any read-ahead performed by the operating system during serial reads.

#### Random Access I/O

Read-ahead is performed normally by most versions of UNIX because of the predominantly serial nature of systems used for development. This may cause throughput problems for business or data-base applications where the I/O is not basically serial. This will normally cause blocks to be read that are not referenced before their space is needed in the buffer cache.

Another factor in random I/O is the effort that must be spent by the operating system in determining the block number corresponding to the location or the current file pointer. This involves searching the inodes, and traversing the tree of indirect inode pointers for very large files.

Program \$5 is an example of a benchmark to measure the efficiency of random access I/O. If the 'rand()' subroutine on the machine returns a value in the range of 1 to (2<sup>31</sup>) - 1 (a 32 bit random number), you should define LONG to truncate the value. This is probably the case for a VAX or other 32 bit integer processor.

## PROGRAM #5

```
#include <stdio.h>
#ifndef BSIZE
#define BSIZE BUFSIZ
#endif

main()
{
    register int fd, i;
    register long offset, rnd;
    char buf[BSIZE];

    if( (fd = open("test",0)) == -1 )
        exit(perror("test"));
    for( i = 0; i < 500; i++ ) {
        rnd = rand();
#ifdef LONG
        rnd &= 0x7fff;
#endif
        offset = ((rnd * (long)500) / (long)0x7fff) * (long)BSIZE;
        lseek(fd,offset,0);
        read(fd,buf,BSIZE);
    }
    close(fd);
}

% cc -o random pgm5.c
or
% cc -DLONG -o random pgm5.c
% create ; time random
```

The difference in throughput between program #4 and program #5 should be due to operating system effort in finding the block for a given file offset, wasted time spent performing read-ahead which is probably never used, and disk arm latency required to seek to and read the needed block.

## PIPES

Pipelines are a fundamental part of the UNIX philosophy of operation. Most filters and other programs are designed to be used in conjunction with pipes to perform more complex jobs. Even the simple job of queueing file to the printer is most often done with a pipeline. Because of this, pipeline efficiency should be a primary concern of users.

Pipes under UNIX are implemented by connecting 2 processes to a memory file. Data is taken from the writer of the pipe and given to the reader. The efficiency with which the data can be moved from the address space of the first process, into the system address space or buffers, and then out to the address space of the second process, and the efficiency of the task switching mechanism, determines the throughput of the pipe. System call

efficiency is also a major consideration if the size of the buffer being read or written is very small, because the number of system calls to process a given amount of data increases so drastically.

Program #6 can be used to calculate pipeline throughput. The program is designed to fork a second process and then write and read a number of blocks through a pipeline. Since most filters use block I/O, this test will also do block I/O. The benchmark should be run with both the native blocksize, the smallest blocksize under UNIX, 512 bytes (if they are different), and with the maximum size of a pipe (typically 4096 or 5120 bytes). This will show the effects of blocking and system call overhead on the pipeline.

#### PROGRAM #6

```
#include <stdio.h>
#ifndef BSIZE
# define BSIZE BUFSIZ
#endif

main()
{
    register int i;
    int fd[2];
    char buf[BSIZE];

    *buf = '1';
    pipe(fd);
    if( (i = fork()) == -1 )
        exit(1);
    if( i == 0 ) {
        close(fd[0]);
        for( ; i < 2000; i++ )
            write(fd[1],buf,sizeof(buf));
    }
    else {
        close(fd[1]);
        while( read(fd[0],buf,sizeof(buf)) != 0 )
            ;
    }
}

% cc -o pipel pgm6.c
% cc -o pipe2 -DBSIZE=4096 pgm6.c
% time pipel ; time pipe2
```

Throughput should be calculated in bytes, by multiplying the block size by the number of blocks and dividing by the time required to run the test. It is important to consider the throughput, and not the elapsed time, as the measure of efficiency, since the larger block size writes are moving considerably more data.

## Results and Conclusions

The results reported below are for 2 very different machines, running 2 very different operating systems. The configurations are given below.

Any single benchmark can be invented to demonstrate the superiority of one machine over another. It is the responsibility of the user to determine whether the benchmarks are fair and representative of the work that the machine will be asked to perform. I hope that the techniques presented here will help users to make their comparisons between values that can be compared, and not between things as distant as apples and oranges.

Machine #1:  
 DEC VAX 11/750  
 2Mb real memory  
 BSD 4.2 UNIX  
 Clock rate: 100Hz  
 Block size - 1024 bytes  
 Price: \$70,000

Machine #2:  
 CIE Systems 680/200 (Motorola 10MHz 68000)  
 512Kb real memory  
 REGULUS (a UNIX look-alike)  
 Clock rate: 16Hz  
 Block size - 512 bytes  
 Price: \$25,000

#### PROGRAM #1 - time command overhead

Machine #1 0.2 real time  
 Machine #2 1.0

#### PROGRAM #2 - System call overhead

	real	overhead (micro-seconds)
Machine #1 (nowork)	0.4	
Machine #1 (work)	4.3	390
Machine #2 (nowork)	1.0	
Machine #2 (work)	4.0	300

#### PROGRAM #3 - Sequential file writes

	real	Throughput (blocks/sec)
Machine #1 (create)	5.3	94.33
Machine #2 (create)	12.0	41.67
Machine #1 (write)	6.4	78.13
Machine #2 (write)	10.0	50.00

#### PROGRAM #4 - Sequential file reads

	real	Throughput (blocks/sec)
Machine #1 (read)	3.2	156.25
Machine #2 (read)	10.0	50.00

#### PROGRAM #5 - Random file reads

	real	Throughput (blocks/sec)
Machine #1 (random)	9.3	53.76
Machine #2 (random)	23.0	21.74

#### PROGRAM #6 - Pipelines

	real	Throughput (Kbytes/sec)
Machine #1 (512)	19.9	50
Machine #2 (512)	8.0	125
Machine #1 (4096)	49.0	163
Machine #2 (4096)	39.0	205

# Implementing XNS Protocols for 4.2bsd

James O'Toole <james@maryland>  
Chris Torek <chris@maryland>  
Mark Weiser <mark@maryland>

Department of Computer Science  
University of Maryland  
College Park, Maryland 20742

## Abstract

We have implemented the Xerox Networking Systems protocol suite in 4.2BSD UNIX<sup>†</sup> in order to communicate with Xerox workstations which do not implement IP/TCP. In this paper, we discuss the problems we encountered while making the necessary changes to the UNIX kernel. We found a multitude of IP/TCP dependencies in the lowest level UNIX device driver and network interface code, which were eliminated by replacing inline code with protocol specific subroutine calls. Another problem was that 4.2BSD expects all protocols to be connectionless or byte stream, and the XNS stream level is neither. We generalized the socket code to satisfy the additional requirements of the XNS stream protocol. The result of all these changes is not just the ability to handle XNS but a more general network system without IP/TCP dependencies, ready for the next generation of protocols.

Portions of this work were supported by grants from the National Science Foundation, the Air Force Office of Scientific Research, and Xerox Corporation.

---

<sup>†</sup> UNIX is a trademark of AT&T Bell Laboratories

## Implementing XNS Protocols for 4.2bsd

### Introduction

We have implemented the Internet Datagram Protocol (IDP) and the Sequenced Packet Protocol (SPP) of the Xerox Networking Systems (XNS) protocol suite. In June of 1984, Xerox corporation awarded the Computer Science Department thirty Xerox Development Environment (XDE) workstations and associated equipment for research purposes. This equipment was not due to arrive until mid-November; in the meantime, we obtained as much documentation as possible, and read. This quickly became boring, so we started looking around for something to *do!* We implemented the XNS protocols under 4.2BSD UNIX<sup>†</sup> on our department research minicomputers because we needed them to communicate with the Xerox workstations.

The Berkeley UNIX kernel supports two different forms of interprocess communication (IPC): pipes and sockets. Pipes are a standard UNIXism, but sockets are a new form of UNIX IPC. As distributed, 4.2BSD supports three different domains of communication within the socket abstraction: UNIX internal, DOD IP/TCP, and Xerox PUP. The UNIX internal protocols do not provide for communication between processes on separate UNIX systems. The Defense Department protocols IP<sup>1</sup> and TCP<sup>2</sup> are used widely, and are the primary means of intermachine communication in use in the department today. The Xerox PARC Universal Protocol (PUP) is a predecessor to the newer XNS<sup>3</sup> protocols. However, the Xerox workstations do not support any of these protocols, so we were faced with two choices:

1. Implement IP/TCP on the XDE.
2. Implement XNS under 4.2BSD.

We chose to implement XNS under 4.2BSD for a number of reasons:

1. We are very familiar with the 4.2BSD UNIX kernel.
2. The Xerox workstations had not yet arrived when we were ready to begin work.
3. We wanted the UNIX machines to communicate with the workstations as soon as they arrived.

This paper describes some of the changes we chose to make to the standard 4.2BSD socket code in order to permit the addition of the XNS protocols, as well as the implementation details of the XNS protocols themselves.

---

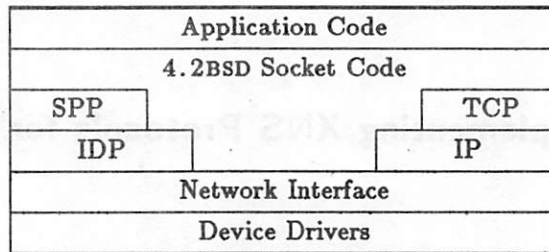
<sup>†</sup> UNIX is a trademark of AT&T Bell Laboratories

<sup>1</sup> See RFC 791 for the complete Internet Protocol specification

<sup>2</sup> See RFC 793 for the complete Transmission Control Protocol specification

<sup>3</sup> See *Internet Transport Protocols* (ISIS 018112), the XNS protocol specification

## 1. Preparing for Multiple Protocol Families



4.2BSD Network Kernel Organization  
Figure 1.

The 4.2BSD socket code is designed around the IP/TCP protocols. Figure 1 shows how the 4.2BSD network kernel is organized, including the XNS protocol implementation we have added. The data structures used for network interfaces contain fields called `if_net` and `if_host`, which are used for speedy access to network and host numbers. However, other (non-IP) addresses have very different formats, so these data structures are too inflexible. Similarly, even general socket code makes assumptions about the internal format of a `struct sockaddr`. Frankly, the problem was that all the code was using `bcmp()`<sup>4</sup> to deal with network addresses. We also modified the code which manipulates `struct sockbufs` to support protocols which differ from those already included in 4.2BSD. All of these changes are internal to the 4.2BSD socket abstraction, and therefore transparent to user code.

### 1.1. Remove Fields from Network Interface Structure

In order to remove these stupid fields (i.e., `if_net`, `if_host`) we had to provide the same address lookup and network number matching functionality as before. So, we rewrote several routines in `if.c` to use address family based comparison routines and use *only* the `if_addr` field. All the code that “knew” about the `if_net` or `if_host` fields had to be changed to use the new versions of these routines.

### 1.2. Address Family Based Address Comparison

Too much socket code made assumptions about the internal format of a `struct sockaddr` in each address family, so we changed this code to use two new routines in the `afswitch` table, based on the address family of the address in question. The new routines in `afswitch` are:

1. `af_addrmatch()`
2. `af_rtnit()`

The `af_addrmatch` routine is a predicate which decides (for a particular address family) whether two `struct sockaddrs` represent the same host. The `af_rtnit` routine is called at interface initialization time to set up a routing table entry in the proper way (for a particular address family). Also, the `af_netmatch` was put to good use by all the code which had been using the `if_net` field of the network interface structure.

### 1.3. Support for Multiple Addresses Per Interface

Since we wanted to use the same Ethernet interface for both our IP/TCP and XNS communications, we needed to support multiple addresses per interface. Then we could get to the *real* work: actually implementing the datagram and packet stream XNS protocols.

To support multiple addresses per interface, we had to hide even more knowledge of `struct ifnet` from most code. That meant we had to modify all the IP/TCP code to use the routines in `if.c` for all access to interface addresses. We further modified all the code in `if.c` to search a list of addresses for each interface, and changed the `if_addr` field in `struct ifnet` into a small array of addresses.

<sup>4</sup> `bcmp()` ≡ assembly code for block memory compare

XNS	OSI	DoD
...	Application	...
Courier	Presentation	...
SPP	Session	TCP
IDP	Transport	IP
	Network	
Ethernet	Data Link	Ethernet
	Physical	

Relationship among XNS, OSI, and IP/TCP  
Figure 2.

#### 1.4. Connectionless, Atomic, and Rights-Based Protocols

The code dealing with socket buffers (send and receive queues) assumed that any protocol was either a byte stream (e.g., TCP, pipes) or a connectionless protocol with optional rights (e.g., UDP). There are protocols which are atomic and also connection based. Such protocols would not pass source addresses with each message. 4.2BSD already includes flags called PR\_ATOMIC, PR\_ADDR, and PR\_RIGHTS; we modified the socket buffer code and the receive system call to use these flags independently. With our changes, a protocol may be atomic without being unreliable, or vice versa. The only remaining restriction is that protocols with source addresses and/or rights must be atomic.

## 2. Actual XNS Protocol Implementation

Figure 2 shows how the IP/TCP and XNS protocol families fit into the the seven level OSI model of the International Standards Organization. There are differences between IP and IDP addressing that cause some problems when trying to splice IDP support into the 4.2BSD socket level interface. The semantic and functional differences between TCP and SPP are so great that providing SPP to the user level through the SOCK\_STREAM type interface is difficult.

### 2.1. Internet Datagram Protocol

Both the Internet Protocol (Defense Department) and the Internet Datagram Protocol (Xerox) are designed to unreliably transport reasonable size pieces of data (*packets*) from one machine to another. Therefore, as one would expect, they are very similar. Both wrap headers around the data to be transported. Both headers provide for source, destination, rudimentary error checking, and some control over who (at the destination machine) receives the data. However, there are a number of big differences between IP and IDP semantics and functionality. Figure 3 shows how these fields are arranged in an IDP header.

IP includes all sorts of doodads for doing prioritized service and security, fragmentation for networks that can't handle large packet sizes, and a multitude of other options (i.e., source routing, recording route taken, packet timestamp, stream identifier). None of these embellishments exist in IDP. In the XNS protocol family, these features are provided by higher level protocols, if at all.

One of the above embellishments that is frequently considered necessary is fragmentation.<sup>5</sup> The IDP strategy for fragmentation is to require the gateways on any network which can't handle the maximum packet size (576 bytes) to implement their own private fragmentation method appropriate to that network. The fragmented packet must, however, leave that network whole once again. To our knowledge, this has never been done, because hardly any networks with such tiny maximum packet sizes use IDP.

The most important difference between IP and IDP is how the sockets are multiplexed across the upper level protocols. In IP, each packet belongs to a particular *protocol*, and the individual upper level protocols (ICMP, UDP, TCP) must implement some kind of socket number in order to separate the traffic in that protocol. In IDP, the source and destination addresses include a *socket* number which accomplishes this separation. Another IDP header field specifies the *packet type*, and this field is interpreted by the process which receives the packet, not by the IDP layer.

<sup>5</sup> Many machines don't implement any of the other IP options.

checksum		
length		
reserved	hopcount	packet type
destination		
network		
destination		
host		
address		
destination socket		
source		
network		
source		
host		
address		
source socket		

**Internet Datagram Protocol Packet Header**

**Figure 3.**

Where the IP code would hand each packet to the upper level handler based on the protocol field of the packet, the IDP code must determine the proper handler for the packet based on the socket field, and the upper layers must interpret the protocol field. To be specific, this means the XNS protocol control block (PCB) data structures comprise one big global namespace. The IP/TCP PCBs, on the other hand, are maintained by the individual protocol code. Also, since the format of the sockets in the IP world is dependent upon the upper level protocol, the IP PCBs are necessarily a hodgepodge of various numbers needed by each protocol.

There are a few other minor differences. IDP packets are checksummed, so the data passed to the upper level is assumed to be correct. In IP, each upper level protocol must perform its own checksum. The internal format of XNS addresses includes both a network number and a host number; the network number is used for routing purposes, and the host number uniquely identifies the machine in question. IP addresses have a more complicated format, and do not uniquely identify a machine. This is one of the reasons `bcmp()` shouldn't be used to compare XNS addresses; the network and socket numbers are not relevant to determining whether two `struct sockaddr_xns` refer to the same machine.

checksum		
length		
reserved	hopcount	packet type = SP
destination		
network		
destination		
host		
address		
destination socket		
source		
network		
source		
host		
address		
source socket		
control		datastream type
source connection id		
destination connection id		
sequence number		
acknowledge number		
allocation number		

Sequenced Packet Protocol Packet Header

Figure 4.

## 2.2. Sequenced Packet Protocol

### 2.2.1. Comparison of TCP and SPP

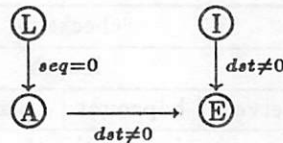
The differences between TCP and SPP are similar to those between IP and IDP. TCP has more options than SPP, while SPP leaves more decisions to the client protocol layer. SPP provides no special acknowledgement sequence to process the closing of a connection; the client protocol must determine when it is safe to shut down a socket. SPP provides more information to the client protocol than TCP. SPP provides end-of-message boundaries and a *datastream type* field, which are interpreted by the client. Figure 4 shows the format of a complete SPP header. The datastream type field is an example of what XNS calls a *bridge* field, that is, a field which is interpreted by the client protocol.<sup>6</sup> The bridge field in IDP is the *packet type*.

Because SPP leaves more of the work to the client layer, implementing it is easier than implementing TCP. However, because SPP provides the extra datastream type information and delimits the end of messages, it does not fit easily into the standard UNIX stream abstraction (*read()* and *write()*). The 4.2BSD socket abstraction also expects protocols to be either

1. Connectionless, usually for unreliable protocols, or
2. Byte Stream, connection based protocols.

The semantics of SPP require that the data packets be delivered to the client separately, and that end-of-message boundaries and the datastream type be communicated as well. Individual protocol requirements such as these are what prompted us to make the changes to the socket level code described in section 1.4.

<sup>6</sup> Courier is the primary client of the SPP protocol.



Homegrown Finite State Automaton for SPP

Figure 5.

### 2.2.2. Formal Protocol Specification

The particular Xerox document we used to develop our implementation of IDP and SPP, *Internet Transport Protocols* (ISIS 018112), contains no formal specification of these protocols. It does provide the format of the headers used, and contains prose descriptions of these protocols in operation. These descriptions suffice to specify IDP, but seem inadequate in the case of SPP; the descriptions are too general, and appear to make optional certain behaviors which are in fact required. We found ourselves asking many questions about the details of SPP.

At any time, an end of an SPP connection (established or not) is in a particular state. When a packet arrives, some action must be taken (i.e., adjust window, acknowledge, drop packet, etc.). A complete specification of a protocol such as SPP would provide an algorithm for determining the proper action to be taken given the current state of the connection and the contents of the incoming packet. In other words, we needed a Finite State Automaton to answer all our questions. Therefore, we constructed our own FSA, illustrated in Figure 5.

Our SPP state machine is remarkably simple. As it turns out, there are only four states for any given connection: LISTEN, INIT\_SENT, ALMOST, and ESTABLISHED.<sup>7</sup> These are represented in Figure 5 as the letters "L", "I", "A", and "E". These states correspond roughly to the TCP states LISTEN, SYN\_SENT, SYN\_RECEIVED, and ESTABLISHED.

In a typical connection setup, the following events should occur:

1. A server enters the LISTEN state and awaits a connection.
2. A client sends a system packet with sequence number zero to the server, and enters the INIT\_SENT state.
3. The server receives the system packet, sends a system packet with sequence number zero in reply, and enters the ALMOST state. At this point, the server may send data, but must not act on any received data (in other words, received data must not be passed up to user code).
4. The client receives the reply and become ESTABLISHED, allowing it to send and receive data.
5. The server receives data (close requests are data, as far as SPP is concerned) from the client, telling it that the client is indeed connected. The server enters the ESTABLISHED state, and may now receive, as well as send, data.

Real networks are not so nice as to deliver packets in order and without loss. Data packets will be retransmitted if they are not acknowledged, but system packets cannot be acknowledged, since they do not consume sequence numbers. Thus data packets are reliably delivered, but system packets are not, and the five-step process above is not sufficient to implement SPP.

With some modifications, however, the procedure outlined above will work. We now describe exactly what action should be taken for an incoming packet in each state. First we make the following definitions:

**Definition.** An *initiating* packet is any packet with sequence number zero and destination connection ID zero (i.e., unspecified).

**Definition.** A *fully specified* packet is one with a nonzero destination connection ID. (This connection ID must match the local connection ID at the receiving machine.)

<sup>7</sup> The Xerox documents use "ESTABLISHED" where we use "ALMOST", and "OPEN" where we use "ESTABLISHED". We prefer our own names.

1. In the LISTEN state (entered when the client opens a passive connection), any initiating packet must be accepted, causing a transition to the ALMOST state, and a system packet with sequence number zero must be sent in reply. Any other packets must be dropped (ignored).
2. In the ALMOST state, any fully specified packet must be accepted, causing a transition to the ESTABLISHED state. Other packets must be dropped.
3. In the INIT\_SENT state (entered when client opens an active connection, at which time an initiating packet is sent), any fully specified packet must be accepted, causing a transition to the ESTABLISHED state. All other packets must be dropped. If no fully specified packet is received within a reasonable time, another system packet with sequence number zero should be sent.

Acceptance of a packet implies that an acknowledgement must be sent (if requested), windows must be updated, and so forth, as outlined in the Xerox XNS documentation.

We claim that these procedures will successfully transfer data even if the the transport layer loses, reorders, or delays packets. Arguments showing the correctness of this kind of reliable transmission protocol abound; we will not repeat them here. However, a connection could remain in the ALMOST state indefinitely if a stray initiating packet were delivered. Aborting such a connection after an arbitrary timeout is unreasonable, because this is a legitimate state: a client might connect to a server but not send data immediately.

### 3. Conclusions About Integration of Multiple Protocols

Implementing another unreliable datagram protocol in the 4.2BSD kernel was not too hard. Implementing a reliable protocol was much harder, because the state information for each connection (e.g., FSA state, sequence numbers, data packets) must be maintained flawlessly. As we write this, the equipment being provided by Xerox Corporation is due to begin arriving next week. Therefore, we have not yet had the opportunity to test our implementations with the actual Xerox implementations. Although our implementation does enable our 4.2BSD machines to communicate, there may be some discrepancies with the standard which will only be resolved in the next few months. In our talk, we hope to have more to report about the success of our implementation.

In order to add multiple protocols to 4.2BSD UNIX cleanly, it is necessary to make small modifications to many portions of the socket code. Some of these modifications require corresponding changes in specialized user level programs such as *netstat*, *ifconfig*, and *routed*. Together, all these changes make our current 4.2BSD kernel very different internally from most other kernels, and any bug fixes or other kernel modifications we receive from other 4.2BSD sites must now be installed with care. We had to modify portions of all code in the *net*, *netinet*, and *vaxif* directories. We estimate that we modified approximately 700 lines of code in these directories, and added 2900 lines of code in the new *netxns* directory. We chose to ignore the PUP implementation distributed with 4.2BSD, because it is superseded by the XNS protocols, and was not in use in the Computer Science Department.

As we modified the 4.2BSD network code, we tried to make it as general as possible without doing much more work than necessary to implement the XNS protocols. Now that we have made these changes, we expect that it will be far easier to add new protocols in the future.

# UNIX KERNEL NETWORKING SUPPORT AND† THE LINC COMMUNICATIONS ARCHITECTURE

Joseph E. Requa

Lawrence Livermore National Laboratory  
P.O. Box 808  
Livermore, CA. 94550  
(415) 422-4036  
requa@lll-crg

## Abstract

This paper describes the UNIX\* kernel networking support developed to allow the LINC communication architecture to be integrated into the Berkeley 4.2 operating system. It also describes the interaction between LINC and UNIX when remote interprocess communication occurs. Features added to UNIX include lightweight system tasks, process interrupt tasks, task scheduling and multithreading within processes.

## Introduction

The LINC communication architecture[1] is based on the International Standards Organization's Open System Interconnect model[2]. It provides location independent, reliable, flow controlled communications over associations between processes. It utilizes the delta-t[3] end to end protocols for network communications. Intramachine transfers are shortcut at the delta-t level for efficiency. LINC provides the underlying communications for the NLTSS[4] distributed operating system being developed at Lawrence Livermore National Laboratory.

The work described in this paper is part of a project to evolve the UNIX operating system into a component of the NLTSS system. The first step in this evolution was to incorporate the LINC communication architecture into the UNIX operating system. LINC has been operational for over a year and is used to support a general file transfer mechanism, a guest file server and several gateways between new and old portions of the Octopus Network.

The general structure of LINC is as follows. The Link level is provided by one or more UNIX device drivers linked to the network

layer by a set of queues and queue handling tasks. The Network layer consists of a set of routines which accept packets, route them to appropriate queues and activate queue handling tasks in the Link and Transport layers. The Transport layer consists of a set of queue handling tasks for input packet handling, a delta-t assurance and flow control task, a set of process interrupt tasks for packet formation and data delivery and a system call task for user services. The Session and Presentation levels are provided as a user library.

Due to the difficulty of handling multiple asynchronous communications in a single threaded program, the application environment provides support for multithreading within a process. The routines for thread management are in a user library, with kernel support for stack overflow management.

At the Application level, processes communicate over associations defined by an address pair, one local and one remote. A single process may have multiple active associations. Two types of information streams are supported, data and control. The data stream uses the alphabet (0,1,B,E,W) where 0 and 1 are the normal data bits and B, E, and W are carried as out of band data. The B delimits the beginning of a data stream. The E delimits the end of a data stream. The W is used to define receiver wakeup points in the data stream.

The control stream is built on the data stream and consists of a set of tokens[5], divided into one or more statements. A token is a quadruple containing a type, a usage, a length and a value. The structure of a token is byte oriented and machine independent to simplify communications between heterogeneous machines. For efficiency, a compressed token format is defined so some tokens can be as small as a single byte. A GO token is used as a statement delimiter.

† Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under contract number W-7405-ENG-48.

\* UNIX is a trademark of AT&T.

Token streams are interpreted using a Monologue Control Record with initialized default token values. When a statement is requested, tokens are parsed and default values replaced until the occurrence of a GO token. After the initial statement is processed, the next statement is requested. This again updates the monologue control record. This mechanism allows commonly used tokens (e.g. an accounting identifier) to be sent only once and held in the monologue control record for multiple usages, minimizing the amount of control information which needs to be transmitted.

Given this communications architecture, the problem addressed by the balance of this paper is how to integrate LINCOS with the UNIX operating system. First, the problem of handling network devices will be described and a suitable solution presented. Second, the modifications to the UNIX kernel necessary for LINCOS support will be described. Third, the requirements and method of introducing multithreading within a UNIX process will be explained. Finally, a concrete example of the interaction between LINCOS and UNIX during interprocess communication will be given.

#### The Problem of Network Devices in UNIX

The UNIX kernel supports two models of I/O device control, dedicated (e.g. tape, terminal) and synchronous shared (e.g. disk). The dedicated model is used for those devices whose hardware characteristics prevent shared usage. The synchronous shared model is used for those devices which can be time multiplexed between processes on a transaction by transaction basis. Network devices do not conform to either of these models, making it difficult to fit networking into the UNIX environment. Adding kernel support for a third model, asynchronous shared, allows UNIX to be extended to a network system in a uniform way. The asynchronous shared model uses addressing information in packet headers to divide a single physical network data stream into multiple independent data channels.

The problem with network devices is that data can arrive spontaneously from other nodes. If the dedicated device model is used, the spontaneous input can be delivered to the device owner for processing. This is a reasonable approach if only process independent functions such as mail or file transport are supported. If, however, general interprocess communications is the goal, either the owning process must provide multiplexing and demultiplexing of communications over the physical channel and use local interprocess communications to complete the transactions or a different device model must be used. If processes are to be allowed direct access to the network device, using the synchronous shared device

model, the system has no way to assure that one process does not receive data directed to another process. Clearly such behavior is unacceptable.

A second problem with network devices is that a complex communication protocol is needed to provide assurance and flow control. To provide uniform I/O access to devices, the system should shield the user processes from the protocols by handling any necessary network protocols.

#### Kernel Additions for Networking

To handle the network device properly, software is needed to perform network protocols and multiplex the device among many users. Two new facilities will allow implementation of such software, a set of "lightweight tasks", which run asynchronously with UNIX processes and "process interrupt handlers" which receive control in process context when activity pertinent to a process occurs. The lightweight tasks provide the asynchronous activity needed to handle the device, independent of which processes are using it. The process interrupt handlers allow asynchronous updating of the process memory when network activity associated with a process occurs.

The UNIX callout queue can be used to implement the lightweight tasks since the callout queue runs on the interrupt stack and is not associated with a particular process, but it is insufficient to schedule the process interrupt handlers. UNIX signals come close to supporting process interrupt handlers, but have the disadvantage that they can not force the process to relinquish control. They are also constrained to be in process memory, not providing simultaneous access to user and kernel memory.

Lawrence Livermore National Laboratory has implemented a set of kernel modifications which provide both lightweight tasks and software interrupt handlers and has developed device multiplexor software to allow processes to be connected to multiple independent network ports utilizing a single underlying network device. Support for multithreading within a UNIX process has also been provided. Based on this experience, we are convinced that if UNIX version 7.2 contained a few simple enhancements to support these features, sites with only binary licenses could add networking devices in a simple and straight forward manner.

The implementation described here is a first attempt at providing the necessary facilities. The next iteration will further minimize the kernel changes needed to support LINCOS. The approach taken is to minimize the changes necessary to the kernel and to build the tasking package alongside existing facilities. The LINCOS communication

architecture is, in turn, built on the tasking primitives.

The first change made was to augment the contents of the callout structure to allow more general tasks to be described. We added process id, task state and task priority fields so both lightweight tasks and software interrupt handlers could be described. A set of flags would also be useful for the general case (e.g. forget/run this task if its associated process dies, this task does/does not need access to its associated process memory, etc.). In retrospect, this was an unnecessary tampering with the existing structure. In the next iteration, task descriptor blocks will be a new structure, separate from the callout queue.

Two task schedulers are provided, one for lightweight tasks and one for process interrupt tasks. Scheduling of the due entries from the light weight task queue is, in theory, subscheduled by priority but the priority aspect has not been needed so currently FIFO scheduling is used. Process interrupt tasks are scheduled in a FIFO manner on a per task basis. The light weight task scheduler is activated by entering it in the callout queue and scheduling a softclock interrupt when appropriate. Scheduling of process interrupt tasks is more complex.

The VAX computer provides hardware support for the scheduling of software interrupt handlers in the form of Asynchronous System Traps (ASTs). This facility, with some collusion by the scheduler, allows process related network tasks to run as interrupt tasks in the user process. UNIX complicates the use of ASTs by storing the process control block in the U area. One can not trigger an AST for a swapped out process by placing the appropriate flags in the process control block.

Our implementation modified the process structure (which does not swap) by adding a state bit flagging LINC's communications usage, and adding a pointer to the head of a list of callout blocks describing pending interrupt tasks. This gives a separate process interrupt task queue for each process. The network communication flag is interrogated when a process exits, to determine if network cleanup is required.

The basic task scheduler (SWTCH) was modified to deliver a kernel mode AST for each newly scheduled process which has interrupt tasks queued (i.e. the new pointer is not null). No other modifications were made to the standard scheduling mechanisms. A kernel mode AST delivery macro was also implemented to deliver interrupts to the currently running task. The trap handler was modified to give control to the interrupt task scheduler whenever a kernel mode AST fired. The interrupt task scheduler copies and clears the pointer to the pending interrupt task list and then runs the tasks in FIFO order. It then checks to see if this

activity has caused some light weight tasks to be scheduled. If so it calls the light weight task scheduler to allow the tasks to run. It then cycles back to the top to see if more interrupt tasks have been queued. In this manner, it makes sure all tasks (except interrupt tasks for other processes) have been completed before giving control to the process.

#### Application Level Network Support

The higher levels of LINC'S[6] provide a set of primitives which split the usual read and write primitives into two halves. The read primitive is split into give empty buffer and a receive full buffer primitives. Similarly write is split into send full buffer and obtain empty buffer primitives. These primitives are nonblocking. A wait primitive blocks until one or more network activities complete. This set of primitives allows simultaneous handling of multiple communications streams.

Unfortunately, the usual single threaded execution model makes handling multiple communications streams difficult, so we provided a multithreaded execution model. This allows a separate execution thread for each stream. In this model, the wait primitive becomes a null operation and the four primitives replacing read and write become synchronous (and may merge back into read and write as more experience is gained). Note that they are synchronous only with respect to their associated threads. The process does not block unless all threads are blocked.

Stack machines are not designed for multithreaded operation so, until heap machines begin to appear, software support is needed to allow multithreaded operation. Two problems arise, stack allocation and stack accessibility. Once a thread is given control, its use of stack space is potentially unbounded, unless some means of limiting usage is provided. A thread may give pointers into its stack space to other threads or the operating system. If its stack is moved, these pointers are invalid, causing access problems. BBN approached this problem by allocating stacks in P0 space with an extra page, containing zeros, appended to each stack. On return from a thread, the page was checked for garbage and, if any was found, it was assumed to indicate stack overflow.

We adopted the same basic model, but provide kernel support for detecting stack overflow. A kernel call was added to mark a page accessible/inaccessible. A signal was added to indicate an attempt to access a marked page. When a new thread is created, the thread scheduler allocates a stack for it in P0 space. Each time a thread is given control, the last page of its stack is marked inaccessible by the scheduler. When the

thread relinquishes control, the scheduler marks the page accessible again. The thread scheduler always has the new signal enabled, so it can obtain control back if a thread overflows its stack.

Because our intent is to evolve UNIX into a network operating system, maintaining a local UNIX environment for system development, we have not linked the LINCOS communication architecture to the usual open, close, read and write UNIX I/O primitives, or to the Berkeley socket primitives, but instead provided a new system call, `lincs`. This call passes a linked list of data transfer descriptors, called buffer tables, to the Transport layer for processing. This communication mechanism is the same as that used in the NLTS distributed operating system we are developing for the CRAY computers. Using this interface, we can port the machine independent servers, developed for NLTS, to UNIX providing the higher system support for a distributed operating system.

### How Remote Interprocess Communication Works

Let us assume that a process on one machine wants to send a block of data to a process on another machine. Since this is a data exchange, the Presentation layer is bypassed. For the sake of simplifying the description of the exchange, assume that the receiver starts first and both ends want to wait until the transaction is complete before continuing. Furthermore, assume that there are no errors in the transaction. Finally, ignore the Link level response generated by the arrival of an acknowledgement or data packet.

#### Initial Activity on the Receive Side.

An association is first opened, by calling the open primitive in the Session layer. It requires a local address, remote address, direction (send, receive, or both), a type (data or control) and security level and returns an association identifier (AID), used as a handle on the association. The open causes state space, for the association, to be allocated and initialized in the Session layer and an AID to be returned. Because LINCOS uses a connectionless protocol, open has no end to end significance, so no activity takes place in lower layers.

Next the receiver calls the give primitive, passing an AID, a buffer starting address and a buffer length, to tell the Session layer where to place the data when it arrives. The Session layer allocates and builds a buffer table (BT) structure to describe the send. The buffer table contains a local and a remote address, defining the association on which the data is to be sent, a base address and length of the data to be sent, a chain

pointer to the next BT in the list and a set of control bits. Bits of interest in this discussion include 1) a direction bit set to 1/0 for send/receive, 2) an activate bit to indicate Transport action is required, 3) a done bit set when the transaction is complete, 4) a wait bit to indicate that the Transport layer is to put the process to sleep until some buffer table with a wait bit on completes and 5) B, E and W bits. In this case, the activate bit is the only bit set. The address of the completed BT is passed to the Transport layer using the new `lincs` system call.

The Transport layer first looks to see if the association is active. In this case, it will not be, since this is the first usage of this address pair. The Transport layer allocates and initializes an Interface State Record (ISR), in kernel space, to record activity on the association. At this point a transmission window has opened at the receiving end, so this information must be passed to the sending end. The Transport layer allocates space for a window opening acknowledgement packet (ACK), builds a delta-t header and calls the Network layer with a pointer to the packet.

The Network layer parses the destination address to select an output queue for the ACK, places it on the queue and calls the task manager to schedule a lightweight task associated with the queue. The task manager places the task in its active queue and sets a softclock interrupt. The interrupt fires immediately, since the machine is running at low IPL. The softclock routine calls the lightweight task scheduler which in turn calls the output queue handling task in the Link layer. The queue handler then calls the device driver to output the ACK. After the driver has sent the ACK, it returns, the queue handler returns, the task scheduler returns and softclock dismisses the interrupt.

We are now back in the Transport layer, which has completed action on the current BT and, finding no others in the chain, returns to the user. This completes activity on the give primitive.

Now the receiver wants to get the data. To do this, the receive primitive is called. This primitive requires an AID indicating the association and a set of addresses of return arguments describing the location and length of the received data. When the Session layer gets the receive primitive, it checks the done bit in the buffer table and, finding it clear, enables a receive wait condition in the state of the association and returns a "not done" indicator. For asynchronous operation, the receiver could go do other work at this point.

Since the receiver wants to wait, it issues the wait primitive, which is the only blocking primitive. Wait takes an AID as input (0 indicating any association) and an

address to store the wait condition which has been satisfied. It returns the AID for which a wait condition has been satisfied. When the Session layer receives the wait, it finds a receive wait condition enabled for the association and no done bit set in the receive BT. It sets the wait bit in the BT and resubmits the BT to the Transport layer.

The Transport layer, recognizing wait set and done clear in the BT, now goes to sleep, suspending the process. This completes activity on the receiving side until something arrives at the network device. The only network activity which has occurred is the sending of the ACK.

#### Initial Activity on the Send Side.

An input interrupt occurs when the ACK reaches the send side, activating the device driver. The driver allocates packet space, reads the ACK into the packet and calls the Network layer to place the packet in an input queue. The network layer queues the packet and calls the task manager to activate the queue handling task. The task is activated and a softclock interrupt is set, but it does not fire because the machine is at high IPL. When the driver releases its interrupt, the softclock interrupt fires and the queue handler task in the Transport layer gets control. Since there is no ISR for the association, the Transport layer discards the ACK and terminates. If an ISR were present for the association, the send window would have been updated and, if the association were flow control blocked, data sending would begin.

Now the sending process starts, opens an association with the same address pair as the receiver (but with the addresses reversed) and executes a send primitive. It requires an AID, a buffer starting address and buffer length. The session layer allocates and builds a send BT and passes it to the Transport layer. The Transport layer allocates an ISR, allocates packet space, builds a delta-t header and copies the user data into the packet. Since the window opening ACK was discarded, and the association is not flow control blocked, the Transport layer uses a default window size. We will assume the amount of data to be sent is smaller than the default window. The Transport layer calls the delta-t module to update the association state, and then calls the Network layer to route the packet to an output queue and start the queue handler. Since we are at low IPL, the softclock interrupt fires and the queue manager in the Link layer gets control. It, in turn, calls the driver to output the packet and returns. The interrupt is released and the Transport and Session layers return.

The sender executes an obtain primitive, which enables a send wait condition and returns not done. It then executes a wait primitive causing the Transport layer to suspend the process.

#### Final Activity on the Receive Side.

The arrival of the data packet, from the send side, causes the device driver to be activated by an input interrupt. The driver allocates packet space, inputs the delta-t header and data, and calls the delta-t module to verify that the packet lifetime has not expired. If it had, a negative acknowledgement would be generated and the packet discarded. The driver then calls the Network layer to queue the packet and schedule the Transport level queue handler. It finally releases the interrupt, allowing the Transport layer task to be started.

The Transport layer passes the packet to the delta-t module to verify that it is acceptable and then queues it on the ISR for the association. It then calls the task manager to schedule a software interrupt handler to deliver the data. The task manager queues the process interrupt handler to the process structure, and wakes the process, which was sleeping in the Transport layer due to issuing a wait primitive. On return, the Transport layer generates a data accept ACK and queues it to the Link level for sending. It then terminates.

The Link layer is now started as a result of a softclock interrupt and the driver sends the ACK. Some time after the Link layer terminates, the UNIX scheduler selects the receiving process to run and, noting it has a software interrupt task queued, delivers a kernel mode AST to the process. The interrupt task is now entered via trap and the task scheduler. It copies the data from the queued data packet to the receivers memory, deallocates the packet and updates the BT by, among other things, setting the done bit. The Transport level task returns, the task scheduler returns and trap dismisses the AST interrupt. When the AST is dismissed, the previously sleeping Transport layer gets control and returns to the user.

The Session level, on return, examines the state of the association and notes that the BT is now done and there is a receive wait enabled. It turns off the receive wait and returns from the wait call, indicating the AID on which a wait condition has been satisfied and an indicator that a receive is now possible on that association. The application now issues a second receive, getting back a good response. This completes the transaction on the receiving end.

### Final Activity on the Send Side.

Except for a few details (e.g. no ACK is generated, no data is copied, an obtain replaces the receive), the ACK sent to acknowledge the data causes completion on the sending side in the same fashion the data packet caused completion on the receiving side.

### Summary

Three simple extensions to UNIX provide a multithreaded network environment without affecting existing applications or violating the spirit of the UNIX system. They are:

1. Implement light weight tasks to simplify handling of network devices.
2. Implement software interrupt handlers to provide asynchronous handling of network events.
3. Implement stack overflow protection to allow the development of multithreaded applications.

Our ultimate goal is to influence AT&T and other UNIX vendors to provide these facilities as a standard part of the UNIX

operating system so our communications architecture, as well as other networking disciplines can be integrated with UNIX without requiring source code.

### References

- [1] Richard W. Watson, "Requirements and Overview of the Lincs Distributed Operating System Architecture," UCRL-90906.
- [2] ISO, "Data Processing - Open Systems Interconnection - Basic Reference Model," Computer Networks 5 (1981) 81, 118.
- [3] Richard W. Watson, "Delta-t Protocol Specification," UCID 19293.
- [4] James E. Donnelley, "Components of a Network Operating System," Computer Networks 3 (1979) 389.
- [5] James A. Minton, James E. Donnelley, Donna T. Mecozzi and Pierre J. Du Bois "The Syntax and Semantics of a Lincs Control Monologue," UCID 18852, Rev. 2 April 1984.
- [6] David M. Hartwell "LINUX Application - Presentation Users Guide," LLNL Working Document.

# TRANSPARENT INTEGRATION OF UNIX AND MS-DOS

Judi Uttal, Jeff Rothschild, and Charles Kline

Locus Computing Corporation

## ABSTRACT

Today's computing environments typically include more than one type of computer, each running a different operating system. Accessing resources efficiently and conveniently across *heterogeneous* machine or operating system boundaries is critical. The Locus Computing Corporation *bridge* solution applies the concept of *network transparency*. The goal is to provide a general purpose *bridge* between two operating systems in a way that does not sacrifice application software, performance, or simplicity of user interface. This paper describes the concept of transparency across heterogeneous operating systems, as applied to the PC-Interface<sup>TM</sup> architecture. PC-Interface bridges an IBM or IBM compatible personal computer running the MS-DOS<sup>TM</sup> operating system into a computer system running the Unix<sup>TM</sup> operating system.

## Benefits of Bridging Heterogeneous Operating Systems

The wide variety of vendors, machine types, prices, applications, and networks has resulted in an increasing need to interconnect machines running different operating systems. Most users require some facility for sharing data and resources between these diverse environments. The *bridge* concept provides a methodology for permitting users and programs resident on one system to directly access resources from another remote system in the same manner as local resources. Resources include files, devices, and processing (such as pipes between two operating systems). To the user (or application program), these operating system extensions are transparent. Application software will immediately operate with foreign data or peripherals without modification or recompilation.

---

PC-Interface is a trademark of Locus Computing Corporation.

MS-DOS is a trademark of Microsoft Corporation.

Unix is a trademark of Bell Laboratories.

## Transparency as a Design Goal

Most network integration schemes focus on the protocols and error handling facility of the network. Once those details are mapped out, a simple file transfer and user interface is developed. That methodology requires either adapting application programs to understand the new program interface, or adding a presentation and an application layer to the network. A *transparent* architecture, on the other hand, strives to maintain the local interface across system boundaries. Preserving the normal interface allows application programs to run without modification and presents a simple user model that approximates a single machine. The operating system handles all networking support functions, hiding network considerations from users and programmers, unless otherwise requested.

## PC-Interface: An Application of the Bridge Concept

This *transparent* approach was selected for the design of the PC-Interface product, a bridge between MS-DOS and Unix. PC-Interface is a general purpose bridge including remote file, device, and processing (terminal emulation and pipes) support. Within the context of this paper we will focus primarily on the remote file system. To the personal computer user or application, the remote file system, stored on the Unix machine, appears to be a local DOS volume. To the Unix computer, the DOS files are also available as native Unix files. PC users and applications may access all files on the Unix system and continue to access all local volumes. DOS application programs can be stored on and executed from the Unix system.

There are clear benefits resulting from the integration of the Unix and DOS environments. In particular, each becomes more commercially viable when they are melded together. For the PC user, access to the Unix system provides a mature centralized file system, where files can be shared and easily backed up. For Unix users, use of personal computers as front ends offloads the multiuser computer from managing the CPU intensive terminal I/O. In addition, it increases the number of application packages available from a few hundred to over ten thousand.

This paper describes the PC-Interface *bridge* architecture from a top down perspective. We will start with an overview of the local area network. Next we will explore the PC-Interface architecture on both the DOS and Unix systems. To dramatize the architecture, an example will be presented. Finally, we will clarify some of the basic problems encountered when building *bridges*.

## Local Area Network Overview

A local area network (LAN) is used as the communication medium between the personal computers and the Unix system. The protocol used by PC-Interface is network independent and can be used on any network, but it is preferable for the LAN to have data transfer rates greater than 50 k bytes/sec. The LAN can be configured with any number of Unix systems, and personal computers. When a PC-Interface connection is established, PC users can select any one of the active Unix hosts.

The PC-Interface implementation can be viewed as two disjoint modules. The assembly language based PC module is loaded at DOS boot time and remains resident to handle remote file activity. The C language based Unix module is invoked when the PC-Interface connection is established. It maintains the network connection and responds to the personal computer's requests.

## Overview of a Standard DOS Application

To understand PC-Interface, one must first understand the DOS execution environment. A schematic of the standard DOS flow of control is illustrated in Figure A. When an application program wishes to access a resource, such as the disk drive, a series of system calls will be performed. Each system call can be viewed as a transaction. The DOS application will issue a system call. The system call includes various parameters such as, the file name and type of access (read, open, close). DOS takes the system call and performs the logical to physical mapping. The resulting physical disk request, such as read sector number 24, is passed on to the device driver. The device driver, which is typically part of the BIOS, communicates the request to the hardware.

## PC-Interface Relationship to the DOS Environment

The major physical difference between a PC-Interface execution environment and the typical DOS environment is that an additional storage resource is available, the remote Unix file system. Since local logical to physical mappings are normally performed by DOS, PC-Interface intercepts all remote file system requests before DOS sees them. Less sophisticated networking schemes capture the remote file system request at the device driver level. However logical information is lost by the time the device driver level is reached. For true integration into a remote environment, such as Unix, the bridge must occur at the logical I/O level.

The PC Module is co-resident with DOS, being loaded at system boot time as if it were a DOS device driver. After initialization, it modifies DOS so that all file system-related calls are intercepted. As illustrated in Figure B, when a file access request is made by an application, the handler captures that request and determines

whether the resource is local or remote. If the resource is local then the call is immediately passed on to the appropriate DOS routine; if the resource is remote then the request is re-packaged and passed on to the connected Unix system. The PC module then simulates all necessary DOS data structures visible to the user program.

## **The Unix Module**

The Unix module executes as a user program. As shown in Figure C, the Unix module is composed of three pieces: the Connection Server, the Network Map Server, and the PC-Interface Server. The Connection Server acts as a network daemon and is responsible for creating and maintaining the network sessions. The Network Map Server receives user requests for a host directory or map, determines the hosts on the network, and responds with a host directory. The PC-Interface Server responds to the personal computer's file system requests. There are dozens of requests that might be sent to the server-- create a file, delete a file, open, close, sequential read, random read, mkdir, chdir, rmdir, chmod, rename, search, etc. To maintain transparency, the response to a request and the action performed must mimic the DOS environment. This next section will describe the architecture components in more detail.

## **Network Map Server**

Within the PC-Interface local area network any number of Unix systems can be active. When a PC user wants to establish a connection with a Unix system he must first see which Unix systems are active. The Network Map Server running on each Unix system is responsible for maintaining a dynamic list of active Unix systems. The Map Server's communicate via a peer protocol designed to ensure that these lists are kept up to date. The dynamic Map Server is a desirable alternative to a static map maintained by a system administrator, because it is more accurate and does not require system administrative overhead.

When a PC user initiates a login session PC-Interface first sends out a request for a site map. At least one Unix system will respond with a map of all active systems. The map is displayed and the PC User selects the host system.

## **The Connection Server**

The Connection Server is responsible for establishing and maintaining connections with PCs. There is one Connection Server for each Unix host in the network. The address of each Connection Server is included in the site map returned by the Map Server. The Connection Server waits on that address for messages from PC's. When a PC wishes to establish a connection to a Unix system, the PC will send a connection establish request to the selected host. If there are sufficient resources on that host for

another PC-Interface session, the Connection Server will create a PC-Interface Server Process, and will provide the PC with an address or port number which can then be used to communicate with the Server Process. The Server Process will have the same attributes as those that would be assigned if the same user established a session via the standard login procedure. In particular, PC-Interface runs with the real user and effective user-ids; thus providing a high degree of security and eliminating additional system administrator overhead.

The Connection Server is responsible for cleaning up the Server Processes as PCs become disconnected. Periodically, all active PC's running PC-Interface transmit a message to the Connection Server letting the Connection Server know that the PC-Interface session is still active. The Connection Server routinely checks it's records of PC connections. If no communication has been received from a particular PC then the Connection Manager assumes that the PC-Interface session has been terminated. The Connection Server will then signal all the PC-Interface Processes associated with that particular PC, and those processes will clean up and exit.

## PC-Interface Server Process

The PC-Interface Server Process responds to the requests made by a PC-Interface User. One PC-Interface Server Process exists for each PC connection. The types of requests issued by the PC-Interface User closely follow the DOS system calls and include: *create*, *delete*, *open*, *close*, *seq\_read*, *seq\_write*, *ran\_read*, *ran\_write*, *chdir*, *chmod*, etc. The PC-Interface Server issues Unix system calls to perform the functions requested by the DOS system calls.

The PC-Interface Server increases performance by anticipating the next PC request. For example, consider the action of reading a file. This operation is initiated by a series of sequential reads; to minimize the time allotted for each individual read, one can anticipate the activity and perform the operation ahead of time. For example, if a file has been opened and a *seq\_read* has been performed, the Server Process will assume that the PC program is sequentially reading a file, and will package the next record immediately. If the next request has been successfully anticipated, the results can be returned to the PC without delay. In general, the Server has a fairly good success rate, thus performance is increased.

## Terminal Emulation

In addition to its file service capabilities PC-Interface also acts as a filter for a terminal emulation facility. Figure D shows the logical architecture of PC-Interface with terminal emulation support.

Under this new model, the PC-Interface Server receives two types of input: PC-Interface file service requests, and terminal emulation input. PC-Interface file server requests are performed consistently whether or not a terminal emulation session is active. However, terminal emulation inputs are passed through to the Pseudo Terminal Driver (PTY). The PTY directs the input data to the appropriate program or shell. The PTY has the same attributes as a normal local Unix terminal. All output from programs is passed through the PTY driver to the Terminal Output Process. The Terminal Output Process is responsible for packaging the output and sending it over the network.

## A Dramatization of PC-Interface

The following example will show the complete series of activities which occur during a PC-Interface session. The network consists of three Unix systems and 10 Personal Computers, all connected via Ethernet. A PC user, Bob wishes to establish a connection with **Frodo**. The user runs the DOS program login. Login issues a network request asking for a map of the active Unix systems. The Network Map Server on one of the active Unix systems receives that request and transmits a site map.

Receiving the site map, the login program displays the menu of systems to the user. Bob selects **Frodo** and is prompted for a login and password. Bob's login and password are forwarded to **Frodo**'s Connection Server. The Connection Server verifies the account. If the account is valid the Connection Server creates a PC-Interface Server Process on **Frodo**. A message is sent back to the PC indicating that the connection has been made and that the remote drive is available as drive C, the next unassigned drive letter.

Bob can now switch over to the remote drive. Executing a DOS **dir** command, he sees the contents of his Unix home directory **/user/bob**. He can actually move around the entire Unix file system via the DOS **cd** command.

Internally, each command he types results in some set of system calls being issued. Each system call is being intercepted by PC-Interface. Those system calls accessing a remote device are packaged up and transmitted to **Frodo**. The PC-Interface Server Process receives the system call stream and processes it. Each call is translated into a set of Unix system calls. For example, Bob wants to run Wordstar, so he types **ws**. DOS attempts to *exec* the Wordstar program. *exec* issues a *open* for read and a series of *seq\_reads*. Since the file is remote, the commands are packetized and shipped to **Frodo**. Bob's PC-Interface Server Process receives the packet and then performs an *open* for read. Upon receiving the first sequential read, Bob's Server Process will execute a *read*, ship the data back to the PC, and then guessing that the next command will be a sequential read, execute another *read*. The data from the second *read* is buffered in anticipation of the next sequential read.

If all has gone well Wordstar will begin executing. Bob can now edit a file from the remote Unix system. Bob will issue a **d** command, indicating he wishes to edit a document. Wordstar will ask which document and Bob will type in "journal". Wordstar will now issue a *open* for read and write, and a *seq\_read*. PC-Interface will catch these commands and again transmit them to **Frodo**. Bob can now begin editing his journal. No further contact will be made with **Frodo** until Bob is ready to save his "journal".

## Perils of Building Bridges

In general, the difficulty in developing PC-Interface was not in creating the network connection, but rather in hiding the differences between DOS and Unix. These differences must be hidden to the degree that applications, users, and DOS itself, are not aware that they are accessing files managed by a different operating system. While most of the operations (system calls) supported by a system such as DOS have their cognates in the Unix operating system, many of them differ in subtle but significant ways. Other types of problems include differences between the naming conventions used by the two systems, different limits on the number of open files or devices, and dissimilar file protection schemes. For example, there are many legal unix file names which do not fit within the DOS file naming rules, and therefore can not be passed back to DOS applications or commands such as DIR. Rather than ignoring these files, PC-Interface dynamically creates DOS legal "handles", which uniquely identify each file. PC-Interface permits DOS users and applications to operate upon these files using the handle in the same manner as if the file name itself were presented.

Another problem, which PC-Interface had to address is that DOS applications can legally have an almost unlimited number of simultaneous open files. As the PC-Interface Server is in fact a single Unix process, and Unix processes can not have over 20 simultaneous open files, it was necessary to create a cache of open file contexts in order to support all legal DOS applications.

Other problems come from the fact that you are bridging systems which have different definitions for what appear to be similar operations. For example, DOS programs can use a form of the open system call which does not specify an access mode. This form of the open, which we will refer to here as an "unspecified open", has no direct cognate on Unix. If this unspecified open were to be treated as an open for both read and write access, as in fact it is on DOS, many applications would not be affected. However, unlike on DOS, a Unix user does not have uniform access rights across the entire file system. Therefore, an application which might behave correctly when run against files in the user's home directory, could fail when run against files for which the user does not have write access. Of course, those applications which in fact attempt writes to files should fail when run against those files for which the user does not have write access. Unfortunately, many "read only" DOS applications use the unspecified open, and would be unnecessarily prevented from running if the unspecified open were to be treated as an open for read and write. An example of such a program is the DOS

**compare** command. **Compare** issues an unspecified open to each of two files. If the protection modes for one of the files does not allow write access for the user, then the program will fail. Unfortunately this isn't the expected behavior for programs such as **compare**. The way PC-Interface handles unspecified opens is by first opening the file for read access only. Later if the program issues a write system call, then the file is closed and re-opened, this time with both read and write access. In this manner, programs which use "unspecified opens", but do not in fact write to a file, will behave as expected with files for which the user has read but not write access.

## Conclusion

This paper describes an example of a *bridge* architecture between DOS and Unix. Whether the architecture and its implementation are successful can be measured by system performance, functionality, and degree of *transparency*.

Performance of PC-Interface must consider the performance on two independent systems: the actual disk access performance apparent to the personal computer user and the impact of PC-Interface on the Unix system. With suitable network support, the performance of PC applications accessing remote files is superior to that of the same applications accessing files from a local hard disk drive. As the disks on some host systems are considerably faster than the hard disks typically provided with personal computers, performance in some environments substantially exceeds that of a stand alone system. Additionally, file access performance is enhanced by sector buffering and read ahead heuristics, both on the host and on the PC. As a result, PC performance using PC-Interface is consistently superior to use of local disks. This characteristic makes PC-Interface the ideal foundation for a series of diskless workstations. On the Unix system, PC-Interface minimally impacts system performance. Infact, when the PC is used as a front-end processor in a distributed network, PC-Interface can actually improve overall system response time.

The functionality of PC-Interface is complete; the architecture and functionality described in this paper have been implemented, tested, and shipped. The product is currently available from AT&T on the 3B family of Unix systems as well as several other vendors systems.

The degree of *transparency* is a new measure of an old concept, user friendliness. PC-Interface is a truly transparent bridge between DOS and Unix. PC-Interface users see virtually no differences between running DOS on a local disk and running PC-Interface with a remote disk. Almost all DOS applications execute remotely without any modification.

As discussed, PC-Interface is just one example of *bridge* technology. This technology can be applied to many operating system environments. Given the multiple operating system environments of today, *bridge* building is becoming the method of choice for integrating these systems. The general goal should be for *transparent bridges*, thereby eliminating the need to rewrite and or modify the current application software base, and to maintain a simple user model.

## DOS Execution Environment

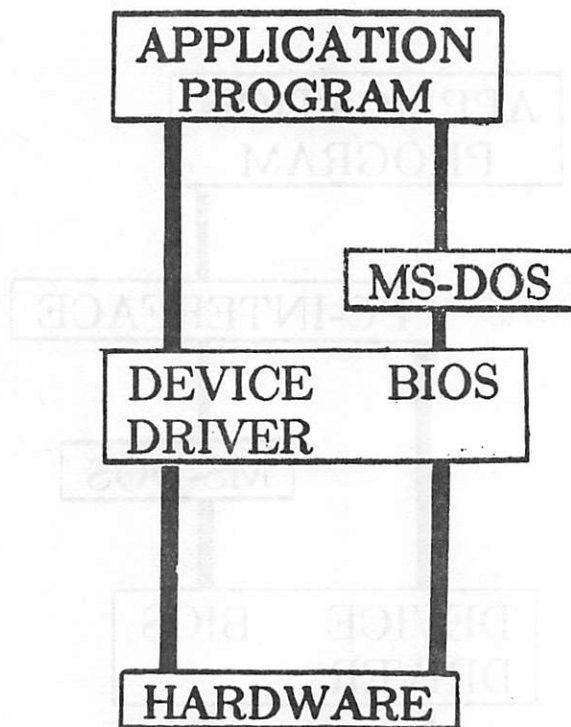


Figure A

## PC-Interface Integration Into the DOS Environment

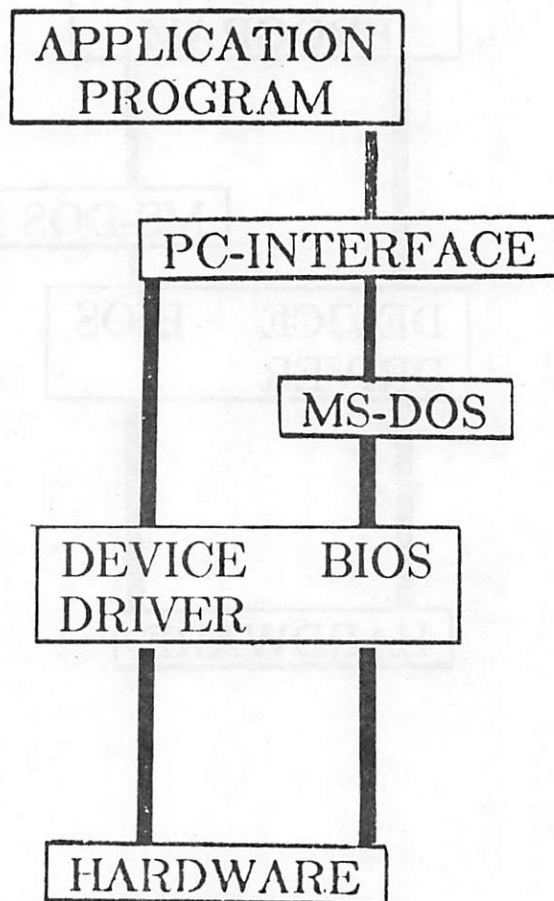


Figure B

The diagram illustrates the PC-Interface Architecture. A vertical line on the left represents the Unix Host. Three horizontal lines branch off to the right from this vertical line, each connecting to a rectangular box. The top box is divided into two sections: 'CONNECTION SERVER' on the left and 'NETWORK MAP SERVER' on the right. The middle and bottom boxes are single sections labeled 'PC-INTERFACE SERVER PROCESS'.

**CONNECTION  
SERVER**

**NETWORK  
MAP  
SERVER**

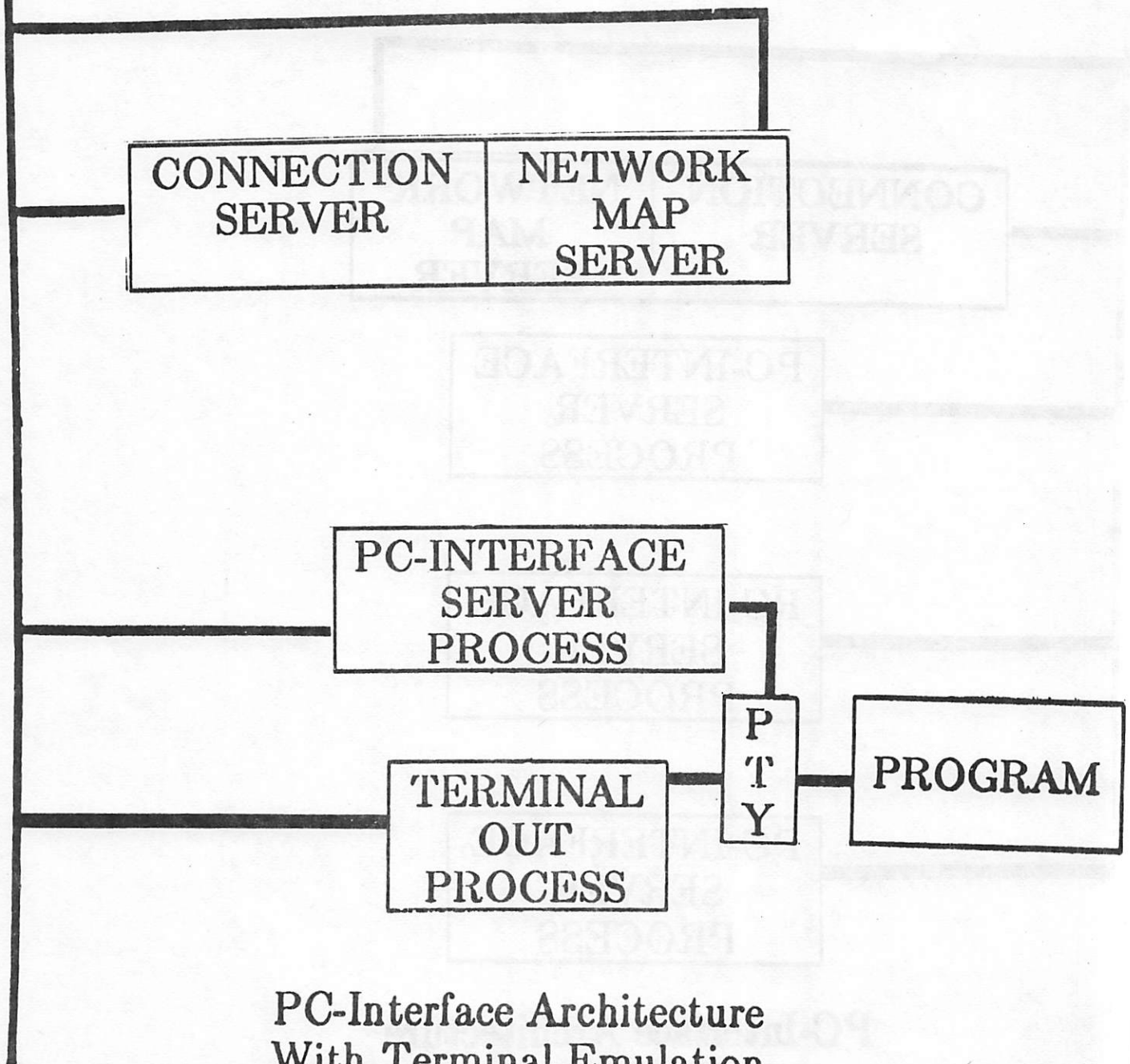
**PC-INTERFACE  
SERVER  
PROCESS**

**PC-INTERFACE  
SERVER  
PROCESS**

**PC-INTERFACE  
SERVER  
PROCESS**

**PC-Interface Architecture**  
**Unix Host**

**Figure C**



PC-Interface Architecture  
With Terminal Emulation  
Unix Host

Figure D

# Overview of the Sun Network File System

Dan Walsh, Bob Lyon, Gary Sager,  
and members of the Sun NFS project:  
J. M. Chang, D. Goldberg, S. Kleiman,  
T. Lyon, R. Sandberg, and P. Weiss

Sun Microsystems, Inc.

*Sun's Network File System (NFS) is a vehicle for sharing file systems in a heterogeneous network of machines, operating systems and networks. The NFS interface is open, and Sun encourages customers, users and other vendors to take advantage of the open interface to extend the richness of the product.*

## 1. Introduction

Sun's Network File System (NFS) permits transparent sharing of file systems in a heterogeneous network of machines, operating systems and networks. The view of the file system seen by a client depends upon mutual agreement of the client and servers; servers supply parts of the file system to the network, and clients have a great deal of freedom in setting up their access. It will usually be most convenient to provide a large UNIX<sup>†</sup> file system to all clients of the network, but service can be tailored to a variety of individual requirements. Clients can make informal arrangements to share files via the NFS without special privilege and without appeal to authority. Because this flexibility can make the maintenance and administration of the system difficult, new tools are provided to make the administrator's job easier.

The NFS was *not* designed by extending the UNIX operating system onto the net. Instead, the NFS is designed to fit into Sun's network services architecture [1]. Thus, the NFS interface is not a step towards a distributed operating system; rather, the NFS interface is designed to allow a variety of machines and/or operating systems to play

the role of client or server. Sun has opened the NFS interface to customers and other vendors in order to encourage development of a rich set of applications, machines and operating systems working together on the network.

## 2. Architecture and Implementation

To add precision to the discussions which follow, it is necessary to define several terms and concepts. First, there is a distinction between the code implementing the operations of a file system and the data making up the file system structure and contents; we refer to the former as the *file system operations* and the latter as the *file system data*. A *server* is a machine that serves resources to the network, and a *client* is a machine that accesses server resources over the network. A machine may be both a server and a client. Finally, a *user* is a person "logged in" at a client, and an *application* is a program or set of programs that run on a client.

In the Sun implementation of the NFS architecture (Figure 1), there are three interfaces to be considered: the operating system interface, the virtual file system node (VNODE) interface, and the network file system (NFS) interface. The operating system interface is important because it is

<sup>†</sup> UNIX is a trademark of Bell Laboratories.

used directly by applications. The UNIX operating system interface has been largely preserved in the Sun implementation of the NFS, thereby insuring compatibility for existing applications.

VNODEs are a re-implementation of UNIX inodes that cleanly separate file system operations from the semantics of their implementation. Above the VNODE interface, the operating system deals in vnodes; below the interface, the file system may or may not implement inodes. The VNODE interface can connect the operating system to a variety of file systems (e.g. 4.2 BSD or DOS); these are designated in Figure 1 as VFS's (virtual file systems). A local VFS connects to file system data on a local device.

The remote VFS defines and implements the NFS interface. The remote VFS uses a remote procedure call (RPC) [2] mechanism; RPC allows communication with remote services in a manner similar to procedure calling mechanisms available in many programming languages. The NFS and RPC "high-level protocols" are described using the external data representation (XDR) [3] package; XDR permits a machine-independent representation and definition of high-level protocols on the network.

Figure 1 shows the flow of a request from a client (on the left) to a collection of file systems. In the case of access through a local VFS, requests are directed to file system data on devices connected to the client machine. In the case of access through a remote VFS, the request is passed through the RPC and XDR layers onto the net. In the current implementation, we use the UDP/IP protocols and the Ethernet. On the server side, requests are passed through the RPC and XDR layers to an NFS server; the server uses the VNODE interface to access one of its local VFSs and service the request. This path is retraced to return results.

It is useful to revisit the above discussion to see how Sun's implementation of the NFS provides five types of transparency:

- 1. File System Type:** VNODE in conjunction with one or more local VFS's (and possibly remote VFS's) permits an operating system (hence client and application) to interface transparently to a variety of file system types.
- 2. File System Location:** Since there is no differentiation between a local and a remote VFS, the location of file system data is transparent.
- 3. Operating System Type:** The RPC mechanism allows interconnection of a variety of operating systems on the network and makes the operating system type of a remote server transparent.
- 4. Machine Type:** The XDR definition facility allows a variety of machines to communicate on the network and makes the machine type of a remote server transparent.
- 5. Network Type:** RPC and XDR can be implemented for a variety of network and internet protocols, thereby making the network type transparent.

Simpler NFS implementations are possible at the expense of some advantages of the Sun version. In particular, a client (or server) may be added to the network by implementing one side of the NFS interface. An advantage of the Sun implementation is that the client and server sides are identical; thus, it is possible for any machine to be client, server or both. Users at client machines with disks can arrange to share via the NFS without having to appeal to a system administrator or configure a different system on their workstation.

### 3. The NFS Interface

As mentioned in the preceding section, a major advantage of the NFS is the ability to mix file systems transparently. In keeping with this, Sun encourages other vendors to develop products to interface with Sun network services. RPC and XDR have been placed in the public domain and serve as a standard for anyone wishing to develop

applications for the network. Furthermore, the NFS interface itself is open and can be used by anyone wishing to implement an NFS client or server for the network.

The NFS interface defines traditional file system operations for reading directories, creating and destroying files, reading and writing files, and reading and setting file attributes. The interface is designed such that file operations address files with an uninterpreted identifier, starting byte address and length in bytes.

Commands are provided for NFS servers to initiate service (*mountd*), to serve a portion of their file system to the network (*exports*), and to retract a portion of their file system from the network (*unexports*). A client "builds" its view of the file systems available on the network with the *mount* command (described below).

The NFS interface is defined such that a server can be *stateless*. This means that a server does not have to remember from one transaction to the next anything about its clients, transactions completed or files operated on. For example, there is no *open* operation, as this would imply state in the server; of course, the UNIX interface uses an *open* operation, but the information in the UNIX operation is remembered by the client for use in later NFS operations.

An interesting problem occurs when a UNIX application *unlinks* an open file. This is done to achieve the effect of a temporary file that is automatically removed when the application terminates. If the file in question is served by the NFS, the *unlink* will remove the file, since the server does not remember that the file is open. Thus, subsequent operations on the file will fail. In order to avoid state on the server, the client operating system detects the situation, renames the file rather than unlinking it, and *unlinks* the file when the application terminates. In certain failure cases, this leaves unwanted "temporary" files on the server; these files are removed as a part of periodic file system maintenance.

Another example of how the NFS provides a friendly interface to UNIX without introducing state is the *mount* command. A UNIX client of the NFS "builds" its view of the file system on its local devices using the *mount* command; thus, it is natural for the UNIX client to initiate its contact with the NFS and build its view of the file system on the network via an extended *mount* command. This *mount* command does not imply state in the server, since it only acquires information for the client to establish contact with a server. The *mount* command may be issued at any time, but is typically executed as a part of client initialization. The corresponding *unmount* command (which replaces the UNIX *umount*) is only an informative message to the server, but it does change state in the client by modifying its view of the file system on the network.

The major advantage of a stateless server is robustness in the face of client, server or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete NFS operations until the server or network is fixed. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a disciplined operations staff and may be running untested systems and/or may be rebooted without warning.

An NFS server can be a client of another NFS server. However, a server will not act as an intermediary between a client and another server. Instead, a client may ask what remote mounts the server has and then attempt to make similar remote mounts. The decision to disallow intermediary servers is based on several factors. First, the existence of an intermediary will impact the performance characteristics of the system; the potential performance implications are so complex that it seems best to require direct communication between a client and server.

Second, the existence of an intermediary complicates access control; it is much simpler to require a client and server to establish direct agreements for service. Finally, disallowing intermediaries prevents cycles in the service arrangements; we prefer this to detection or avoidance schemes.

The NFS currently implements UNIX-style file protection by making use of the authentication mechanisms built into RPC. This retains transparency for clients and applications that make use of UNIX file protection. Although the RPC definition allows other authentication schemes, their use may have adverse effects on transparency.

Although the NFS is UNIX-friendly, it does not support all UNIX file system operations. For example, the UNIX "special file" abstraction of devices is not supported for remote file systems because it is felt that the interface to devices would greatly complicate the NFS interface; instead, devices are implemented in a local */dev* VFS. Other incompatibilities are due to the fact that NFS servers are stateless. For example, file locking and guaranteed APPEND\_MODE are not supported in the remote case.

Our decision to omit certain features from the NFS is motivated by a desire to preserve the stateless implementation of servers and to define a simple, general interface to be implemented and used by a wide variety of customers. The availability of open RPC and NFS interfaces means that customers and users who need stateful or complex features can implement them "beside" or "within" the NFS. Sun is considering implementation of a set of tools for use by applications that need file or record locking, replicated data, or other features implying state and/or distributed synchronization; however, these will not be made part of the base NFS definition.

## 4. Example

Figure 2 shows a possible file system view as seen by three machines. The server machine (bottom) has exported the */usr2* and */usr* trees with the commands:

```
exportfs -a /usr2
exportfs -a /usr
```

The "-a" option indicates that the directories can be mounted by any client.

The clients (called "blue" and "red") have each mounted subtrees with the commands:

```
mount -t nfs server:/usr2 /usr2
mount -t nfs server:/usr/src /usr/src
```

The "-t nfs" option is used to indicate that the type of file system being mounted is remote over the network; the location of the file system is indicated by prepending the server's machine name and a colon to the name of a directory in an exported tree. Note that the clients selectively mount */usr/src* rather than all of */usr*. With this arrangement, all three machines share the trees named */usr2* and */usr/src*.

Figure 2 also shows a sharing arrangement between the blue and red machines. The blue machine has a directory on a local disk with the name */usr/proj*, and has exported it with the command:

```
exportfs -a /usr/proj
```

The red machine has mounted the subtree with the command:

```
mount -t nfs blue:/usr/proj /usr/proj
```

Thus, the blue and red machines share the */usr/proj* directory, while the server is unaware of the arrangement. If users of the blue machine feel that it is important to keep the sharing more private, the *exportfs* command can restrict the set of machines permitted to mount the subtree.

## 5. Performance

Our performance goal is to achieve the same throughput as measured on a previous release of the system that used the network only as a disk (and thus did not permit sharing). Measurements are taken on a well-defined configuration and set of benchmarks. Current indications are that we will attain that goal.

The Sun implementation of the NFS has a number of performance enhancements, such as "fast paths" to eliminate the work done for high-runner operations, asynchronous service of multiple requests, caching of disk blocks, and asynchronous read-ahead and write-behind. The fact that caching and read-ahead occur on both the client and the server effectively increases the cache size and read-ahead distance. Caching and read-ahead do not add state to the server; nothing (except performance) is lost if cached information is thrown away. In the case of write-behind, both the client and server attempt to flush critical information to disk whenever necessary to reduce the impact of an unanticipated failure; clients do not free write-behind blocks until the server verifies that the data is written.

The flexibility of the NFS allows configuration for a variety of cost and performance trade-offs. For example, configuring servers with large, high-performance disks and clients with no disks may yield better performance at lower cost than having many many machines with small, inexpensive disks. Furthermore, it is possible to distribute the file system data across many servers and get the added benefit of multiprocessing without losing transparency. In the case of read-only files, copies can be kept on several servers to avoid bottlenecks to the information. In more complex situations, trade-offs can be made between keeping portions of the file system data local or remote.

## 6. The Yellow Pages

The Yellow Pages (YP) [4] is an independent service from the NFS; we include this discussion because the YP plays an important role in initialization and administration of the NFS as installed at Sun.

From the point of view of the servers and clients, the YP is a centralized read-only database. For a client, this means that an application's access to the data served by the YP is independent of the relative locations of the client and server.

It is important to note that the YP provides a client access to data without recourse to the file system. This fact allows greater generality in the initialization of clients by allowing them to access information needed to mount file systems without requiring them to mount the filesystem containing a file with that information.

The YP is a collection of cooperating server processes that use a simple discipline to distribute data among themselves. Thus, the servers share the load of providing access to data and the failure of a server need not disable the network. The YP does not implement a true distributed database: for every relation in the database, one YP server is designated to control the update of data for the entire collection of YP servers. Thus, the administration of an entire network of servers and clients is done from a single point of contact. Should the control server fail, an alternate server can be designated as the control. The policy for distributing changes through the network yields a weak form of consistency: the databases across the network will be consistent after a "reasonable" time has elapsed. A system administrator can choose to have changes distributed periodically according to a schedule, or can cause them to propagate immediately.

The most obvious use of the YP is for administration of */etc/passwd*. Since the NFS uses a UNIX protection scheme across the network, it is advantageous to have a

common `/etc/passwd` database for the servers and clients on the network. The YP allows a single point of administration and gives all servers and clients access to a recent version of the data, whether or not it is held locally. To install the YP version of `/etc/passwd`, existing applications were not changed, they were simply relinked with library routines that know about the YP service. Conventions have been added to library routines that access `/etc/passwd` to allow each client to administer its own local subset of `/etc/passwd`; the local subset modifies the client's view of the system version. Thus, a client is not forced to completely bypass the system administrator in order to accomplish a small amount of personalization.

The YP interface is implemented using RPC and XDR, so the service is available to non-UNIX operating systems and non-Sun machines. YP servers do not interpret data in the databases, so it is possible for new databases to take advantage of the YP service without modifying the servers.

## 7. Conclusion

The NFS is designed to provide file system service in a heterogeneous network of machines and networks. Sun has implemented the NFS interface for the Sun Workstation and 4.2 BSD UNIX operating system. New tools are provided to aid in the administration of an NFS network. The NFS interface is designed to encourage further development of the nodes and the network; Sun plans to use it as a basis for further product development and has opened the interface to customers and other vendors in order to encourage an atmosphere of mutually beneficial product development.

At this time (December 18, 1984), the NFS has been used for serious work for over two months; steady improvements in robustness and performance have been made based on experience gained. During the first weeks, crashes brought to light many bugs; however, no files were lost. We do not yet have MTBF data, as we tend to boot new

versions before failures occur (every 3 to 5 days). Presently, we are converting all internal development to use the NFS, and will collect reliability data based on general use within Sun. The NFS will be shipped to customers in Spring, 1985.

Sun is collaborating with several other vendors to make the NFS available as part of their product line.

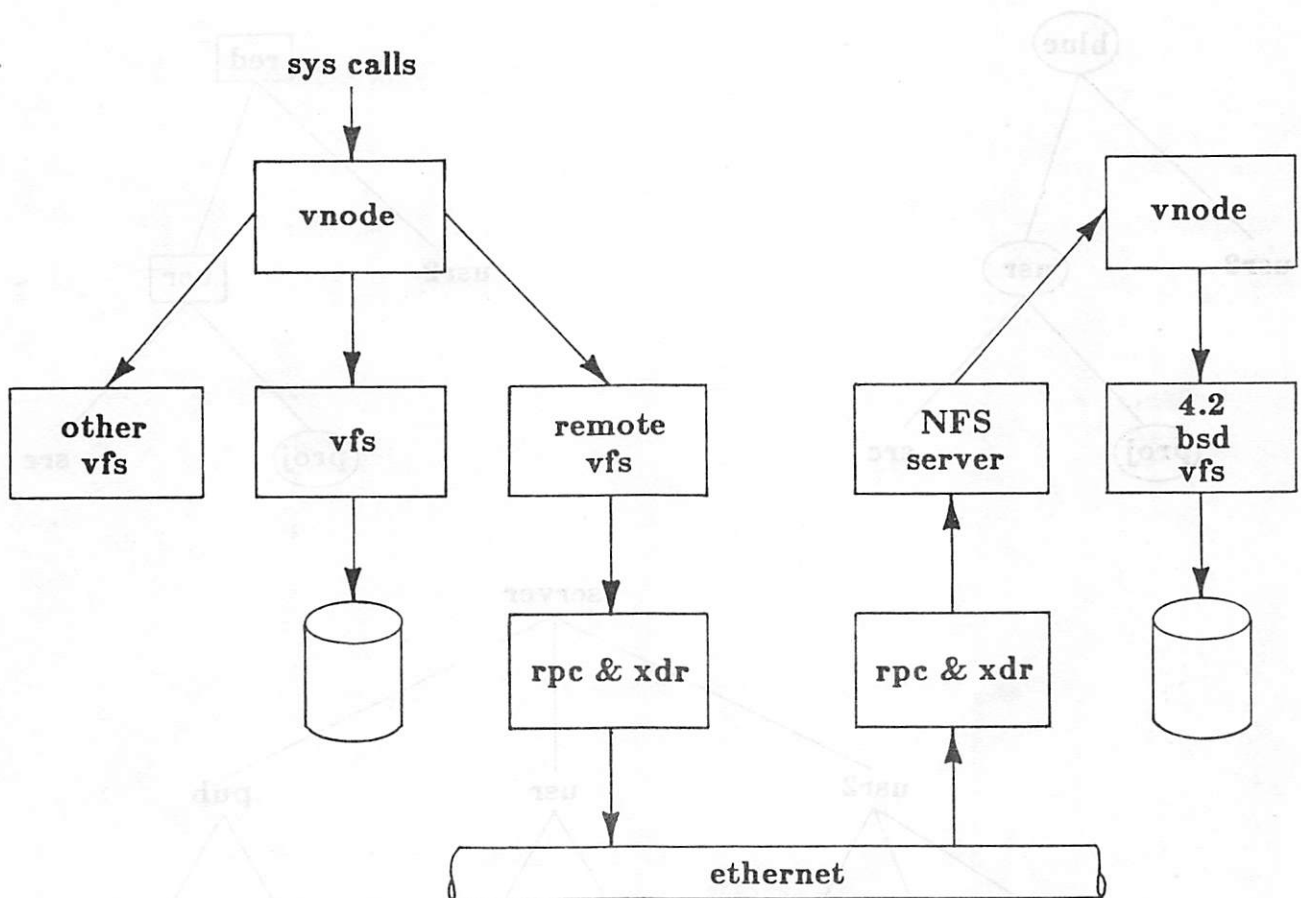
## 8. Acknowledgements

The network services architecture and the NFS were championed at Sun by Bill Joy. Bill Shannon and other Sun employees have provided a great deal of valuable advice to the NFS project.

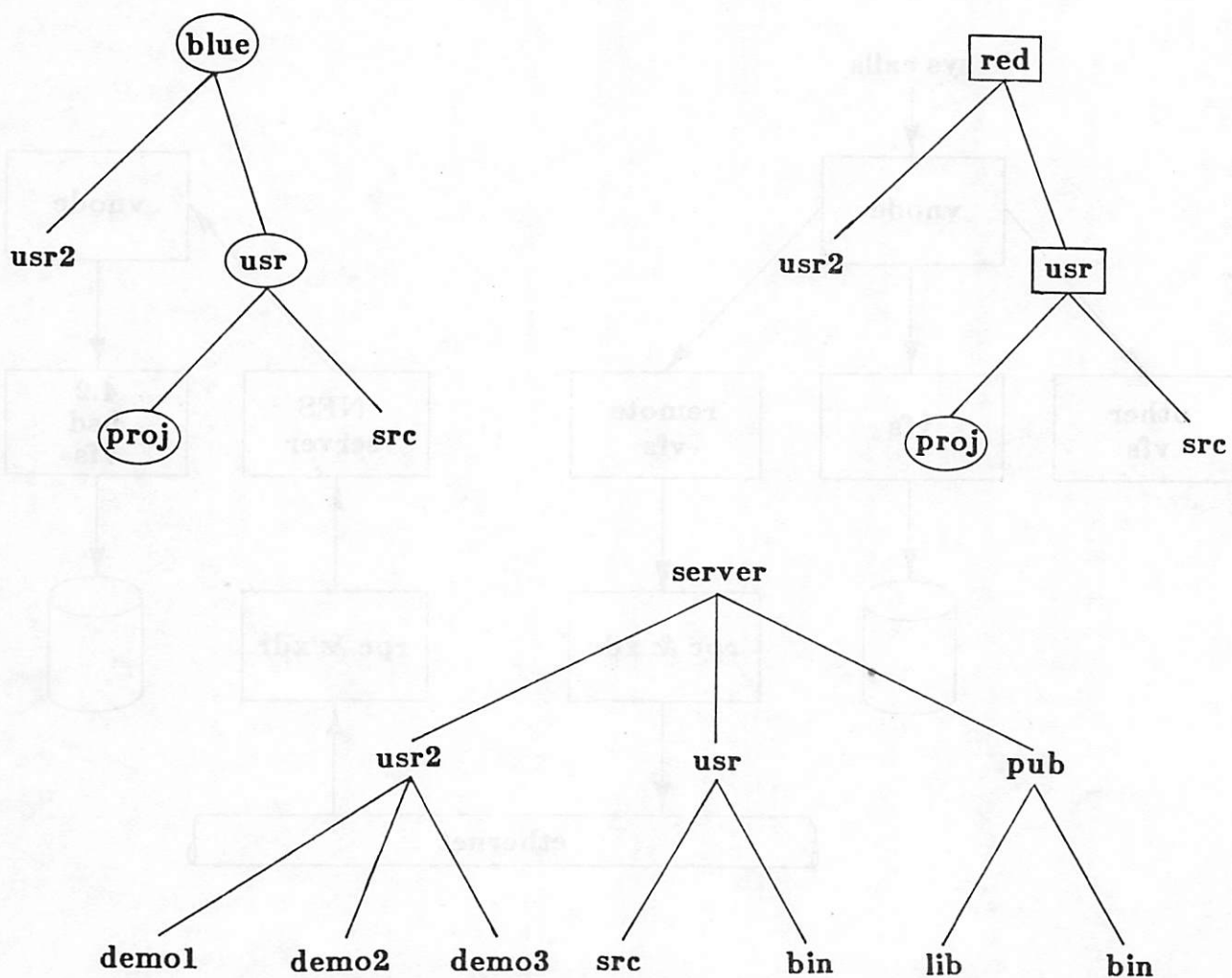
## 9. Bibliography

- [1] Joy, W. N. *The UNIX System in the Laboratory*, UNIX/WORLD, vol. 1, no. 4, 1984, pp. 34-38.
- [2] Remote Procedure Call Reference Manual, Sun Microsystems, Inc.
- [3] External Data Representation Reference Manual, Sun Microsystems, Inc.
- [4] Yellow Pages Reference Manual, Sun Microsystems, Inc.

**Figure 1: NFS Architecture and Implementation**



**Figure 2: Example NFS Use**



# Latent Source Bugs and UNIX<sup>†</sup> System Portability

Alan Filipiski  
SW Engineer, UNIX group  
Motorola Microsystems  
2900 S. Diablo Way  
Tempe, AZ 85282

{allegro | ihnp4} ! sftig ! mot ! al  
{ihnp4 | seismo} ! ut-sally ! oakhill ! mot ! al

## ABSTRACT

*The type of C language portability problem discussed in this paper is the "latent source bug", i.e. the coding error which is innocuous on one machine but is deadly on another. An example of such a bug is a structure element which is defined as a char in one place and char \* in another. On some machines, structure padding will cause each definition to allocate 4 bytes for the element, whereas on machines which require less liberal padding, the structures will be misaligned, causing serious problems.*

*This paper provides some insight into this type of problem by presenting a discussion of some of the latent-bug portability problems found in UNIX System V code during Motorola's System V port. Five different types of such problems are discussed in which (incorrect) C code worked correctly in the VAX System V environment but failed on a Motorola M68000 based machine. The symptoms of these bugs were sometimes bizarre or amusing, as when the output from nroff(1) consisted of characters shifted one position in the collating sequence, e.g., "Hello World" became "Ifmmp Xpsme".*

## 1. CONTEXT: THE SYSTEM V/68<sup>TM</sup> PORT

In Early 1983, we at Motorola Microsystems entered into a contractual agreement with AT&T Inc. wherein we would port the UNIX System V operating system to run on our M68000-based computer products. The result would be a "generic" M68000 System V port (called SYSTEM V/68) which would run on our EXORmacs<sup>®</sup> and VME/10<sup>TM</sup> computer systems and would be available for OEM ports to other M68000-based machines.

<sup>†</sup> UNIX is a trademark of AT&T Bell Labs

Several other major semiconductor manufacturers entered into similar agreements with AT&T at about the same time. Although ports to microprocessors had been performed before, this marked the first time that AT&T had ever contracted with others to perform officially sanctioned ports of the UNIX Operating System. We had two options for execution of the port— we could either contract the work out to an independent UNIX system porting house, or we could invest in the development of in-house expertise and perform the port with our own personnel. We chose the latter course. (Despite persistent reports to the contrary in the trade magazines.)

Most of the programming work for the port was performed on a VAX 11/780; our procedure was to modify UNIX System V VAX source code and cross-compile it on the VAX for down-loading to the M68000-based EXORmacs computer system. By April 1984, our port was officially validated and accepted by AT&T and was made available to our customers.

During this port work, many different types of impediments to portability had to be addressed. Besides “substantive” issues such memory management, device drivers, efficient use of hardware, etc., many C language portability issues emerged. Sometimes, as in the case of the infamous “hibyte-lobyte” issue these were dependent upon interaction with the architecture of our target machine. Other times we needed to address purely compiler-dependent issues. This paper considers one particular type of language portability problem, namely the “latent source-code bug”. This situation occurs when there is some bug, or logical inconsistency, in the original source code which was “luckily” symptomless on the original machine but harmful in the environment of the new compiler or machine architecture.

## 2. A CATALOG OF LATENT BUGS

This section is an itemization, in retrospect, of some of the more interesting “latent-bug” problems we encountered in porting the half-million lines of UNIX SYSTEM V operating system source code to the M68000. It should be a useful guide for anyone who needs to do similar work.

*2.1 The a0/d0 Problem* An interesting latent bug was the “a0/d0” function return problem. On the VAX, all function return values in C are left in the r0 register by the compiler. Hence it does not matter on the VAX whether a function is declared, say, to return an `int` or a pointer, the returned value will always be in r0. The M68000 compiler, however, returned `ints` in the d0 data register and pointers in the a0 address register. Mismatched function declaration/invocations thus caused the return of garbage for us. For example, the procedure `malloc()` is a frequently used library function which returns a pointer (`char *`) to some free space. If it is invoked by a program which declares it as or let it default to

```
int malloc();
```

instead of

```
char *malloc();
```

Then, in the case of the M68000, `malloc` put the returned value ( a pointer) in the address register a0 but the calling program looked for it in d0 where a returned `int` would belong. The calling program therefore used garbage from d0 instead of the pointer in a0. (Actually it was later found that the values remaining in d0 were typically numbers like ‘0’, ‘1’ or ‘2’, which were left over from previous temporary calculations and function calls). Because of the explicit pointer cast in the assignment

```
p = (char *)malloc(sizeof(buf));
```

there was no compiler warning. By using these small garbage numbers instead of pointers to legitimate free space, we were effectively using the text (instruction) space at the beginning of user memory as a free memory pool! The cause of the bug was only discovered after Read-Only text segments were implemented and we experienced segment violations trying to write to write-protected segments. It is perhaps a testimony to the robustness of UNIX system that programs actually ran fairly well in this state.

This problem was masked on the VAX because a single register (r0) was used for all returns.

Utilities rendered non-portable by this bug included `fsck(1M)`, `dc(1)`, `file(1)`, and `getty(1M)`.

`Diff3(1)` had a similar problem, involving an undefined return value, which prevented it from working correctly on the M68000.

*2.2 The Padding Problem* Here was another example of a bug that happened to be symptomless on the VAX but was deadly on the M68000. In the `nroff(1)` code, there was a structure element declared as a character in one place and as a pointer in another. On the VAX, this was innocuous, since the field was padded to a four-byte boundary on that machine and the character took up as much space as the pointer. The M68000 SGS II C compiler, however, pads to two-byte boundaries. The `char` type is allocated two bytes and the pointer is allocated four, so that all items in a data table following the structure in question were off by two bytes. The symptom produced by this bug was rather amusing. The characters output from `nroff(1)` were shifted one position in the ASCII collating sequence. For example, "F" became "G", "P" became "Q", and "Hello World" became "Ifmmp Xpsme".

The following is a simplified description of what happened: Two header files contained declarations of what were supposed to be the same structure (struct `ttable`). The last item in the structure, however, was declared as

```
char zzz
```

in one place and

```
char *zzz
```

in another. In the course of the execution of the `roff(1)` code, `ttable` is written to an intermediate data file, followed by a string translation table. Later, all this data is read back in. The size of the reads and writes is determined by `sizeof(ttable)`. On the VAX, because of structure padding, `sizeof(ttable)` was 1000 in each case and there was no inconsistency. On the M68000, however, the version of `ttable` with the pointer was 1000 bytes long, while the version with the `char` was 998 bytes long. This inconsistency caused the string table which followed this structure to be shifted two bytes, causing the strange symptom.

The `nroff(1)/troff(1)` system contains about 180 files and working on this problem consumed at least ten man days just before we were scheduled to send SYSTEM V/68 to AT&T for approval. This bug nearly delayed submission of our ported system. What made this difficult to find was that it was a bug which was present in the VAX code but was symptomless there because of a coincidence. On the M68000, that coincidence no longer held.

*2.3 The Flexname Problem* Syntactically, our SGS II cross compiler for the M68000 differed in several ways from the resident C compiler on the VAX. For example, our new compiler supported arbitrary-length identifiers, with all characters considered significant (flexnames),

whereas the resident VAX compiler considered two identifier names the same if they agreed in the first 7 characters. One might think that the new feature, since, in a sense subsuming the old, would not cause any portability problems. However, here is what happened: The old code would contain two variable references such as "baseaddress" and "baseaddr" with the intent that they refer to the same thing. Well, under the new compiler, they would no longer be considered the same and problems would arise. I have always hated the compiler "feature" which allows one to use long identifiers but only attached significance to the first few. This confirmed my dislike.

Utilities affected by this problem were uucp(1C), sdb(1), expr(1), wall(1M), mail(1), and newform(1).

**2.4 The Null Pointer Ref Problem** In C, a string variable is a pointer. It points to a sequence of bytes terminated by a 0 byte. A variable representing a null string is thus a pointer to a zero byte, e.g.

```
s[0] = '\0';
```

makes the value of s a null string. The assignment

```
s = 0;
```

is something quite different. It makes the string pointer s point to memory location 0. The value of the string itself is whatever happens to be at memory location 0. If we try to print this string out with the statement

```
printf("%s", s);
```

we would get a string of ASCII characters representing whatever was at location 0. The string would go on until a null character (0) were encountered. The UNIX System V code contained something similar to this. The variable s was declared as

```
static char *s;
```

and then, under a certain chain of circumstances, was printed out before anything was assigned to it. (Under C, statics are initialized at 0, so this is equivalent to assigning 0 to s.) Well, what is at location zero? Location zero is always the first word of the text space of a UNIX program. On the VAX, the first byte of text space is a register mask which always happens to be 0. Therefore, it just so happens that an uninitialized static string is null and will not appear as garbage when printed out. This is however, a bad coding practice. This is another example of a hidden bug, waiting to cause problems when the code is transported to another machine. If one prints s out on the M68000, there happens to be no register mask and one gets garbage in the form of instructions interpreted as ASCII characters. This affected the utility awk(1).

**2.5 The hbyte-lobyte Problem** This is the classic UNIX system portability problem. It refers to the difference in significance ordering of bytes within short (16-bit) or long (32-bit) words. On DEC and some other equipment, the first (lowest-addressed) byte of a multi-byte integer is the least significant, while on most other machines, including the M68000, the first byte is the most significant. In most ordinary C programming, this difference is of no consequence. It is possible, however, to write C code that has different effects in these two environments. For example, if we declare the variable x as a union

```
union { char c[4];
        int i; } x;
```

and execute the assignment `x.i=1`, then on the VAX, `x.c[0]` will be equal to 1 and `x.c[1]`, `x.c[2]`, and `x.c[3]` will be equal to zero. On the EXORmacs, however, `x.c[3]` will be equal to 1 and `x.c[0]`, `x.c[1]`, and `x.c[2]` will be equal to zero. In terms of bits, `x` looks like this on the VAX:

```
00000001 00000000 00000000 00000000
```

and like this on the M68000-based machine:

```
00000000 00000000 00000000 00000001
```

There are also a few other ways to invoke this machine dependency in C, such as writing data to a file in one binary format and reading it back in another. Shifts are not a problem. Regardless of the internal representation, compilers insure that, for example

```
x >> 10
```

is, for any positive integer `x`, the same as

```
x/1024
```

In the case of some utilities such as `sdb(1)`, whose code is already complicated by being reworked by several generations of programmers, it took a lot of study to determine exactly how to fix the byte-order dependencies. The writing of byte-order dependent code is just a bad coding practice. It can and should be avoided by all C programmers.

Another ramification of this problem was that any files containing multi-byte entities such as `shorts` or `ints` required selective byte-swapping before being downloaded to be run on the M68000 target. This was not too difficult for COFF (Common Object File Format) files, since we had a utility to do exactly this, but some data files used by the utilities were more troublesome. We finally decided to create many of them on the target machine instead of downloading them from the host.

We now give a specific example of how the hi-byte-lo-byte problem masked a bug in UNIX source code so that the code worked correctly on the VAX, but failed when we transported it to the M68000. (Non-essential details have been changed, however to protect the source-licensee.) The problem occurred in the kernel in the implementation of the system call `mount(2)`. This system call takes three arguments: two pointers to `char` and an `int`. The pointers refer to the name of the special file and the name of the directory on which it is to be mounted, respectively. The `int` is a "read-only" flag which can be 0 or 1. This argument list is placed as three four-byte quantities into a field of the user structure. The address of this field is `u.u_ap`. The readonly flag is subsequently referenced, however, as

```
((struct a *)u.u_ap)->ronly
```

Where `a` is defined as

```

struct a {
    char *fspec;
    char *freg;
    char ronly;
};

```

The situation is similar to the union described above. On the VAX, assigning a one into the 4-bit int which is the third argument is the same as assigning a 1 into the `char` `ronly`. On the M68000, however, `ronly` is always zero whether the third argument is set to one or not. The latent bug is that `a` should have been defined as

```

struct a {
    char *fspec;
    char *freg;
    int ronly;
};

```

With this fix the code works correctly for both machines.

Besides the kernel, the utilities affected by hi-byte-lo-byte problems included `prs(1)`, `tplot(1G)`, `fsdb(1M)`, `unpack(1)`, and `sdb(1)`.

### 3. CONCLUSIONS

C is widely acknowledged to be one of the most portable of the Algol-like block-structured programming languages. There are a number of reasons for this. One of these, paradoxically, is the relative low level of C. Almost any low-level manipulation can be implemented efficiently as a C function, so there is not a strong tendency to extend the language. Another reason for the *de facto* standardization of C is that for a long time there was essentially one textbook on C and one large company promoting C. This is obviously changing, and we can only hope that the newly formed ANSI/IEEE efforts toward the standardization of C will be as successful. I do not want to give the impression with this paper that I think C has substantial portability problems. The fact that we went through several hundred thousand lines of code and found only a few portability problems attributable to C or C coding practices is amazing.

Many of the C portability problems we found were really in the form of latent bugs, that is, bugs which were masked on the original system by some coincidence and were revealed only when the code was compiled and tested on the target machine. These include the `a0/d0` problem, the structure padding problem, the flexname problem, and the null pointer reference problem. In some cases, however, such as the hi-byte-lo-byte issue, the problem was usually not covert bugs but simply poor coding practices.

# The Clipboard Data Interchange Facility

Robert T. Nicholson  
Sydis, Inc.

## Introduction

As UNIX systems proliferate, the long-awaited application software is beginning to appear. Unfortunately, UNIX applications may require a great deal of work in order to compete with current desktop computer applications, many of which are sophisticated second or third generation products. One feature that is rapidly becoming a requirement is data integration.

Data integration takes two forms in the applications world. The first is horizontal data integration, in which data is transferred from one format, such as a spreadsheet, to another, such as a word processing document. This type of integration is becoming familiar to many users today through the Apple Macintosh "cut and paste" facility. The second type of integration, vertical data integration, is used for transferring data between two similar contexts; for example, between a spreadsheet program running on a PC, and a spreadsheet program running on a UNIX system.

Early data interchange facilities were largely limited to simple ASCII tables. As user requirements grow more sophisticated, however, data interchange facilities must be expanded to meet their needs. Some of these "next generation" data interchange requirements are listed in the following section.

## Requirements

- 1) Semantic content must be included with the data. When copying data from one spreadsheet to another, for example, it must be possible to transfer not only the current value of a field, but the formula that was used to compute the value. In other words, vertical data interchange must preserve not only the fields, but the relationships among the fields as well.
- 2) Formatting information must be included with the data. In a spreadsheet or database, each field, row, or column may have particular formatting information associated with it. For example, a field might be a monetary unit, displayed with a floating dollar sign. If this field is copied to another spreadsheet, the formatting instructions should be preserved, so that even if the value later changes, it will still be displayed with a floating dollar sign.
- 3) Standard data representations must be used to avoid international barriers. In other words, dates, times, currencies, et cetera, should be transferred in a canonical form, to allow programs running in different countries to exchange information. As an example of this, suppose that a database containing several dates is prepared in the United States.

The dates might be entered and displayed in the form "mm/dd/yy." However, if the data from this database is sent by electronic mail to a European user, it might be displayed in the form "dd.mm.yy."

- 4) Non-ASCII information must be supported. Current applications, particularly communications and office automation systems, display and manipulate a variety of data types, included image (facsimile), graphics, and digitized voice. The data interchange format should be general enough to support these types, as well as others that might arise in the future.
- 5) Compound objects must be supported. The data types listed above are often combined into "compound objects;" for example, a document may contain a graph and several voice annotations. Since a user may want to "cut" a section from the document and paste it into another, the data interchange format must be able to deal with mixed data types.
- 6) A standard means must be provided to identify the program (and version) that produced the data interchange file(s). In this way, programs that are specifically designed to "cooperate" can pass highly specific information, while less tightly coupled programs can accept and use only the information that they are prepared to handle.

We will discuss such "levels of communication" in a later section of this paper.

### The Clipboard Facility

The Clipboard data interchange format has been developed to address these and other requirements. "Clipboards" are implemented as UNIX directories, allowing a single clipboard to contain multiple types of data in individual files. The core of the clipboard is a single data file that contains header information, an ASCII representation of a table of data, and several types of auxiliary information (described below). Additional files may be contained in the clipboard for bit-map image, graphics, or voice data.

Specific features of the clipboard format have been designed to address the requirements defined in the previous section, as explained below:

- 1) Semantic content is preserved by associating a required "type" field, and an optional "auxiliary" field, with each ASCII data field. The type field defines the basic semantic type of the data item, ie, integer, string, date, et cetera. The optional auxiliary field may be used by cooperating applications to contain such information as formulas or field relationships.

- 2) Formatting information is preserved between cooperating applications by using optional auxiliary elements associated with each data row and data column.
- 3) International data formats are insured by defining canonical representations for date, time, and currency data elements.
- 4) Non-ASCII data is contained in separate files within the clipboard directory. Fields that "contain" non-ASCII data are stored as pointers to the files where the data actually resides. Several standard types of such data are defined, including VDI graphics and FAX. Other vendor-specific types may also be defined and may be passed between cooperating programs.
- 5) Compound objects are naturally supported using the multiple-file format described above.
- 6) The header portion of the standard clipboard data file contains space for program name and version identification, to aid programs in determining their degree of compatibility.

#### Levels of Communication

Point (6), above, is intended to allow varying levels of cooperation among applications using the clipboard data interchange format. These levels are roughly defined as follows:

**Minimum support:** All applications that support the clipboard format should be able to produce and accept ASCII representations of any of the following types: integer, floating point, string, date, time, and currency. The most typical use for minimum support is in horizontal integration; for example, copying a table from a spreadsheet or database into a document. In addition, rows and columns must be preserved where applicable (for example, copying a 3 row by 4 column table from a spreadsheet to a database should result in a 3x4 table in the database).

Applications that support the clipboard format at a minimum level should extract the information that they can make use of, and must not be affected by additional information that may be contained in the clipboard directory.

**General support:** At this level, applications are expected to accept and support not only the ASCII form, but the canonical form of each of the data types listed above; eg, dates must be accepted as dates and displayed in the appropriate local formats. Also, as above, applications must simply ignore information that they are not able to process.

**Cooperating application support:** This level is intended to be used by developers of integrated sets of applications. At this level, programs may "recognize" one another by the program identification information included in the clipboard header. They

may then make use of private conventions for passing additional information in the field, row, and column auxiliary data areas.

## The Clipboard Library

Given the potential complexity of a clipboard, support at even the minimum level could require a fair amount of programming effort. Further, small differences in interpretation of the format could result in application programs that were unable to exchange data.

For this reason, the clipboard facility is defined as a library of C language routines, rather than as a data format per se. The library includes the following routines:

`clip_create` - creates (and opens) a clipboard directory.

`clip_open` - opens an existing clipboard directory.

`clip_close` - closes an open clipboard directory.

`clip_put_header` - writes a "header" structure to an open clipboard. Headers contain the following information:

- o The name of the program that produced the clipboard.
- o The revision level of the program.
- o The name of the vendor that developed the program.
- o The title of the clipboard.
- o The revision level of the clipboard,
- o Auxiliary information (comments, formats, etc.)
- o The number of columns (fields per row) in the clipboard.
- o Type, width, and auxiliary information for each column.

`clip_get_header` - reads the clipboard header structure from an open clipboard.

`clip_put_row` - writes a data row structure to an open clipboard. If any field(s) in the row contain non-ASCII data pointers, the data is copied from the files where it resides into files within the clipboard. Data rows contain:

- o Auxiliary information for the row.
- o ASCII values and auxiliary information for each field.

`clip_get_row` - read a row from an open clipboard. If any fields within the row contain non-ASCII data, the pointers to the data files within the clipboard are returned to the caller.

Routines are also provided to pack a data row into a single ASCII string, to unpack an ASCII string into a data row (breaking it up into fields at boundaries specified in a header structure), and routines to convert data such as dates and times into and out of their canonical forms.

The clipboard library is currently being used by its developers to implement a fully integrated set of office applications supporting text, data, voice, and graphics. Beginning in April of this year, it will be distributed for a nominal copying charge to qualified UNIX developers, in order to encourage its adoption as a useful data interchange standard.

## ABSTRACT

UNIX is a general-purpose operating system for multi-user environments. It is designed to be portable and to support a wide range of hardware and software configurations. The system is based on a set of standard interfaces and a set of standard libraries. The system is designed to be easy to use and to be easy to extend.

The system is designed to be easy to use and to be easy to extend. It is designed to be portable and to support a wide range of hardware and software configurations. The system is based on a set of standard interfaces and a set of standard libraries.

Conclusion and summary of the system. The system is designed to be easy to use and to be easy to extend. It is designed to be portable and to support a wide range of hardware and software configurations.

## Introduction

Many people think that UNIX is a simple system. It is not. It is a complex system that has been designed to be easy to use and to be easy to extend. It is designed to be portable and to support a wide range of hardware and software configurations.

What is UNIX? UNIX is a general-purpose operating system for multi-user environments. It is designed to be portable and to support a wide range of hardware and software configurations. The system is based on a set of standard interfaces and a set of standard libraries.

Let me tell you a little bit about UNIX. UNIX is a general-purpose operating system for multi-user environments. It is designed to be portable and to support a wide range of hardware and software configurations. The system is based on a set of standard interfaces and a set of standard libraries.

Can't Happen  
or  
/\* NOTREACHED \*/  
or  
Real Programs Dump Core

Ian Darwin  
Geoff Collyer

University of Toronto Computing Services  
255 Huron Street - UTCS  
Toronto, Ontario M5S 1A1  
lhnp4!utcs!{ian, geoff}

ABSTRACT

UNIX<sup>†</sup> programmers too often fail to check for failure of system calls or functions, taking the familiar teen-age attitude that "it can't happen to me (or my program)". This paper will attempt to convince its audience to take prophylactic measures. Those who take such measures will be healthier - and less prone to surprises - than those who don't take such measures.

In the tradition of the classic *Elements of Programming Style* some real-world programs will be criticised publicly. Actual production (or *soi-disant* production) UNIX programs and subsystems will be examined. Each of these "provides one or more lessons in style." We present both before and after versions of most code fragments.

Come on out and see if we abuse one of your programs - or your programming style!

Introduction

Many people think that errors can only happen to "the other guy." Unfortunately, current versions of UNIX do not provide a system call to tell if you're running as the other guy... But I hear that Dennis is working on it!

Why is net.bugs among the most active newgroups? Why is the USENIX 84.n tape filled with nothing but 4.2 bugs and fixes? Why does *each* release of System N claim to *fix* hundreds of new bugs? Clearly something is rotten in the state of software. One thing that we see wrong is the attitude that "errors won't happen to me, so I don't need to check for them."

Let me tell you briefly (I promise!) A Tale of Two Systems, the UNIX and the Emveeous. "There dwelt in the land of New Jersey the UNIX, a fair maid whom savants travelled far to admire." The Emveeous was envious of the UNIX for her natural grace; the UNIX however, spurned the Emveeous, thinking him a coarse, vulgar fellow. One day her suspicions were confirmed when she saw that he had more manuals listing his errors and sins than she had manuals describing her entire life. But as the UNIX grew into middle age and got busy, she became careless, and made many mistakes, and forgot to check these errors. And the scribes duly observed these errors, and duly recorded them. And the UNIX died old and unhappy, for she saw

-----  
<sup>†</sup> UNIX is a trademark of Bell Laboratories.

in her final hour that her error messages manual had grown malignantly to become large, larger than that of that old simpleton, the Emveeous.

Is there a moral in this sad tale? If there is, I believe it is this: If you want a system that forces you to do things its way, that handholds and spoonfeeds you, that spends a third of its resources checking for errors you might have made, that spews myriad messages on your terminal at random intervals, that sings you a sad song when you leave out a comma, in short, "If you want MVS, you know where to find it."

What I plan to do is to present some guidelines for safe, surprise-free programming. Taking after *The Elements of Programming Style*, we'll present real-world examples, suggest improvements and draw conclusions.

The examples are drawn from our own experience in maintaining code on the dozen UNIX systems we currently maintain on four different computer architectures. The examples are taken from several UNIXes to which we have access (some source and some binary), and from public domain code.

We hope that those whose code we have criticised will take it as constructive criticism rather than as personal criticism. Our aim is to improve, not to insult.

## 1. Examples

Pontification without proof is pointless. Here are some examples of ineffective code and how it can be improved.

### 1.1. The art of thinking

The art of thinking (before you code) often seems a lost art. Such safeguards as validating the input before you read it and keeping the user interface constant from one command to the next, are good things whose time has not (we hope) truly passed.

#### 1.1.1. Check the input?

There are times when it's easy to check the input for certain obvious errors. Many programs now check that their input file is not a directory; this is probably a good thing. Much work remains to be done in the area of input validation. Here's a simple example from a binary-only system (UniSoft System V):

```
$ tc /tmp filep
t8d{ @T@t@T@T@mm@lg@7~@Ho@M@L@L@H@la@6v@h@'@L@N@V@H@s5r@g@4y@q@G@
c@3d~@u@f@M@L@L@H@n2x@m@1@w@I@I@C@Hj@0{I@P@H@P@;;lu@Nf@H@I@I@L@
H@;;da@M/r@H
@I@I@L@Hd@.u@lk@9mk?RfRGRGRIRM7l @;
$ p
```

On a Tek terminal, the results would be less spectacular but no less erroneous. Consider another example:

```
$ tc pascalprog.p
```

The results would be similar to that shown.

The only valid input to this program is a file created by the old (non-di) *troff*. These files invariably start with an initialise command, which has the octal value 0100.

Note also that the second argument is ignored when no error message has been printed. The program behaves as if it 'thinks' that all is well, but produces voluminous trash, in the presence of a single typographical error.

```
$ tc /tmp filep
tc: /tmp: not troff output
tc: file: cannot open (no such file or directory)
$
```

How little work it would be to check for this error, and how much more pleasant it would make life, is something to cogitate on.

---

### Test input for plausibility and validity

---

#### 1.1.2. Directories are fun

Directories can be a lot of fun when you read them into a program which expects a data file. Many places check, but many more do not. Here is how 4.2 Berkmail works.

```
$ Mail -f /usr/spool/mail
"/usr/spool/mail": 0 messages [Read only]
& h
No applicable messages
& x
$
```

How it might behave:

```
$ Mail -f /usr/spool/mail
Mail: "/usr/spool/mail" is a directory!
$
```

#### 1.1.3. Don't change the interface

USG systems (PWB, System III, System V) come with *labelit* and its brethren.

```
$ labelit /dev/rgmc0a gmc0a root
Current fsname: ROOT, Current volname: gmc0a, Blocks: 13566, Inodes: 1904
FS Units: 1Kb, Date last mounted: Thu Jan 3 20:09:25 1985
NEW fsname = gmc0a, NEW volname = root -- DEL if wrong!! <type DEL>

$
```

There are a couple of problems with the example above: a strange interface, and no feedback where feedback is called for.

There is something quite backwards about this program's behaviour. The program tells you what it's going to do, says 'DEL if wrong!!', waits 10 seconds, then goes ahead and does it!

Everything else in UNIXdom either assumes that you know what you're doing, or asks with some user-friendly prompt like

```
last chance before scribbling on /dev/....
```

What logic can there be for the decision to make this program use a whole new method of interaction? The rationale may be that USG systems are designed for large DP shops, with Cobol and Operators. These programs are designed to 'simplify' operations, or to run 'unattended' (that's when the system is allowed to continue running while the operator is asleep in front of the console). While I sympathise with the problem, I don't agree with the solution.

Scenario: you are converting from v7 to System III. You've just typed a command

```
volcopy /dev/hp1b /dev/hp0c
```

that will copy the distribution over top of what you've been working on all night, instead of vice versa. The phone rings (or, worse, your manager walks in and just *has* to talk right this instant). Any other UNIX utility will wait until you get off the phone. Volcopy, however, will wait ten seconds. Well, I hope you got it right.

Just as a person who holds another at gunpoint assumes full moral responsibility for the actions of his victim, the programmer who *forces* the user to interact with the program (as opposed to typing a command and having it done) takes on responsibility for the user's actions. The least this program could do, having honked at me for three lines of drivel, would be to tell me whether it went ahead and did the change!! The system I was on was quite busy, and several seconds went by before I typed the DEL (what does the program do if I have my INTR key set to CTRL/C and my erase key is DEL?), and several seconds more before the prompt came back. Was the change done? Quite honestly, at the time of writing, I do not know. Glad it was a labeling job, not a disk-to-disk volume copy!

Here's how it could have been:

```
$ labelit /dev/rroot root
labelit: special file /dev/rroot not found
$ labelit /dev/rgmc0a gmc0a root
Current fsname: ROOT, Current volname: gmc0a, Blocks: 13566, Inodes: 1904
About to change fsname to gmc0a, volname to root - type a 'y' to continue:y
$
```

In this case no confirmation is necessary; if I type y it will do it, if not, not. This is repeatable and predictable, so no feedback is needed, although it would not be out of line (given the importance of the operation) to printf "done" after the write.

The conventional UNIX interface is widely used and understood. The next major interface will probably be something like what the Blit (I'm sorry, the 5620 DMD) provides. Let's not go half-way in the interim.

---

Use a consistent type of dialogue.

---

## 1.2. Read the fine manual (rtfm)

UNIX systems do not have a twelve foot shelf of manuals so there's really no excuse for writing reams of C code before you've read most of the manual set. But people do it.

### 1.2.1. Signals - to catch or not to catch

Some programmers have still not read *UNIX Programming* by Kernighan and Ritchie in Volume 2 of the UNIX Programmer's Manual. Their programs catch signals such as interrupt and quit like this:

```
#include <signal.h>
extern int onintr();

signal(SIGINT, onintr);
```

When such a program is run in the background, it will continue to catch signals generated from the keyboard.

```
#include <signal.h>
extern int onintr();

if (signal(SIGINT, SIG_IGN) != SIG_IGN) /* iff not ignoring interrupts */
    (void) signal(SIGINT, onintr);      /* then catch them */
```

Programs which catch keyboard signals even in the background can make their users wary of ever typing interrupt or quit (or hanging up their connection), since the users aren't sure whether or not some background program will spring to life and erroneously catch the resulting signal. This is arguably a design flaw in the UNIX signal mechanism. Perhaps UNIX should ignore attempts to catch signals which were being ignored when the current program was *execed*. This would make the use of backgrounding with *&* and *nohup* fool-proof. However, UNIX is what it is.

---

Don't blindly catch signals

---

### 1.2.2. System Calls

In what program do we find the following code sequence:

```
open("/", 0);
dup(0);
dup(0);
```

V7 *init*, alas. There are two other occurrences, in which the opens are "open(tty, 2);" and "open(ctty, 2);". The author *knew*, by god, that those system calls could *never* fail. As a result, when the kernel file or inode tables fill, *init* fails to re-populate some terminals with *init* children and thus *getty*'s. Thus those terminals will never inherit *init*'s until the next crash or reboot. We know: the file and inode tables aren't supposed to fill, thus it **can't happen**. A common counter-argument is that in such a case there is nothing sensible to be done. Yet a moment's thought often reveals a better alternative than failing to check. In this case, since the code in question is running in a child of *init*, *init* can sleep briefly and try again if a system call such as *dup* fails or if an *open* fails due to resource exhaustion.

### 1.2.3. /dev/kmem - open sewer or open sore?

A common disease in programs written at Berkeley is to open */dev/kmem* and grub the load averages out by the dirtiest means possible. The follow is from the 4.2BSD *sendmail* source, *conf.c*, slightly reformatted for brevity.

```

#include <nlist.h>
struct    nlist NI[] = {
#define    X_AVENRUN    0
    { "_avenrun" }, { 0 },
};

getla()
{
    static int kmem = -1;
    double avenrun[3];

    /* kmem opened here and nlist called for /vmunix with NI */
    (void) lseek(kmem, (long) NI[X_AVENRUN].n_value, 0);
    (void) read(kmem, avenrun, sizeof(avenrun));
    return ((int) (avenrun[0] + 0.5));
}

```

(void) read should be avoided. The running kernel may not be /vmunix and the read may not return as many bytes as expected, leaving trash in avenrun.

```

#include <nlist.h>
#include <sys/types.h>
struct    nlist NI[] = {
#define    X_AVENRUN    0
    { "_avenrun" }, { 0 },
};

getla()
{
    static int kmem = -1;
    double avenrun[3];
    extern off_t lseek();

    /* kmem opened here and nlist called for /vmunix with NI */
    if (lseek(kmem, (long) NI[X_AVENRUN].n_value, 0) < 0 ||
        read(kmem, avenrun, sizeof(avenrun)) != sizeof(avenrun))
        return -1; /* can't be a valid load average */
    return ((int) (avenrun[0] + 0.5));
}

```

Data that you "know" are there, won't be.

-----  
 Assume that system calls will fail capriciously.  
 -----

#### 1.2.4. Files will always open and they never need to be closed

B news demonstrates most of the bugs discussed in this paper. This is just one small example: inews.c often fails to check that fopen succeeded and fails to close open FILES.

```

actfp = fopen(ACTIVE, "r+");
for(;;) {
    fpos = ftell(actfp);
    if (fgets(afile, sizeof afile, actfp) == NULL) {
        unlock();
        return FALSE;          /* No such newsgroup locally */
    }
    ...
    fclose(actfp);
}

```

Stdio tends to be unamused when handed null pointers, as it would be if ACTIVE could not be opened for reading and writing for any number of reasons: no such file, no permission, full i-node table, full file table, etc. If fgets encounters end of file, this code will return, leaving ACTIVE open.

```

actfp = fopen(ACTIVE, "r+");
if (actfp == NULL)
    xerror("Cannot update %s\n", ACTIVE);
for(;;) {
    fpos = ftell(actfp);
    if (fgets(afile, sizeof afile, actfp) == NULL) {
        unlock();
        fclose(actfp);
        return FALSE;          /* No such newsgroup locally */
    }
    ...
    fclose(actfp);
}

```

Files will sometimes be unopenable for reasons you don't foresee when coding; it is as well to be prepared for such possibilities, however unlikely you (erroneously) believe them to be.

-----  
 Don't assume God likes you: open and fopen will fail.  
 -----

### 1.2.5. System calls never fail in my programs

The ultimate arrogance is to assume that the system calls and functions invoked by one's program can never fail. The following is from 4.2BSD, `/usr/src/lib/libc/net/ruserpass.c` and is so spectacularly bad that we shall return to it.

```

ruserpass(host, aname, apass)
    char *host, **aname, **apass;
{
    reenv(host, aname, apass);
    if (*aname == 0 || *apass == 0)
        rnetrc(host, aname, apass);
    if (*aname == 0) {
        char *myname = getlogin();
        *aname = malloc(16);
        printf("Name (%s:%s): ", host, myname);
        fflush(stdout);
        if (read(2, *aname, 16) <= 0)
            ...
    }
}

```

This code knows that *getlogin* and *malloc* can never, ever fail. Unfortunately it is wrong. Redirecting all three standard file descriptors away from any terminal (e.g. as *nohup*(1) does) will make *getlogin* fail consistently. When this happens, *ruserpass* will cheerfully dereference *myname*, which is now a null pointer, possibly causing a core dump.

*Malloc* seldom fails on working programs in 4.2BSD on a VAX. As a result, programmers develop the nasty habit of failing to check for *malloc* failing. In addition to making their own debugging harder (since *malloc* can fail when you are debugging a program, even on 4.2), this causes endless grief for those using non-paging UNIXes.

---

All the world's NOT a VAX.

---

```
ruserpass(host, aname, apass)
char *host, **aname, **apass;
{
    renv(host, aname, apass);
    if (*aname == 0 || *apass == 0)
        rnetrc(host, aname, apass);
    if (*aname == 0) {
        char *myname = getlogin();
        if (myname == NULL)
            myname = "unknown";
        *aname = malloc(16);
        if (*aname == NULL)
            error("can't allocate memory for password", (char *)NULL);
        printf("Name (%s:%s): ", host, myname);
        fflush(stdout);
        if (read(2, *aname, 16) <= 0)
            ...
    }
}
```

This version will behave sanely when functions or system calls fail,  
unlike the original.

---

Do something sensible when system calls or functions fail

---

### 1.3. Don't Reinvent the Flat Tire

The C libraries contain many, many valuable routines. So does */bin*. Why reinvent them? I know not, but people do.

#### 1.3.1. Use existing tools

What does this do?

```

if (freopen("/usr/lib/whatis", "r", stdin) == NULL) {
    perror("/usr/lib/whatis");
    exit (1);
}
gotit = calloc(1, (unsigned) blklen((int *)argv));
while (fgets(buf, sizeof buf, stdin) != NULL)
    for (vp = argv; *vp; vp++)
        if (match(buf, *vp)) {
            printf("%s", buf);
            gotit[vp - argv] = 1;
            for (vp++; *vp; vp++)
                if (match(buf, *vp))
                    gotit[vp - argv] = 1;
            break;
        }
for (vp = argv; *vp; vp++)
    if (gotit[vp - argv] == 0)
        printf("%s: nothing appropriate\n", *vp);

```

What would happen if the *calloc* call failed?

The Berkeley system contains the command *apropos* used to find manual keywords. While its name might more appropriately have been *findman*, the program *knows* that it is to be called as 'apropos', because it looks at `argv[0]`. It is glued into the source for the *man* command. At any rate, the code attempts to reimplement *grep*, and does so in a way that is possibly correct but inarguably slow. Here is our *findman*, which runs about three times as fast as *apropos*.

```

#!/bin/sh

# findman - find manual pages given topic(s)

PATH=/bin:/usr/bin:/usr/ucb ; export PATH
INDEX=/usr/lib/whatis

for f
do
    grep $f $INDEX || echo 'basename $0: nothing appropriate for $f'
done

```

There has been some debate on the net recently (it is a recurring topic) about replacing C programs with shell files and *vice versa*. Our criteria state that a program must do one function that is unique, and do it well, to be a C program. When you reinvent a shell file to be a C program, you lose the benefit of years of tuning ('hacking?') which has gone into the underlying tool, in this case *grep*.

---

Don't reinvent the flat tire

---

### 1.3.2. Parsing Program Arguments (*argv* scanning)

What does this do on your 4.2 system?

```
/bin/mail -r
```

There are billions of different ways of parsing *argv*, the list of command line arguments passed to a C program. The problem is that they are all different. The USG long ago recognised this as a

serious problem, and implemented *getopt(3)* to handle the problem. Several public domain versions of this routine have been posted to net.sources; it really should be in every C library on every UNIX system in the world. If it's not on your system, add it. And use it.

Use of *getopt(3)* is a standard at our installation. We have in a file called */usr/pub/template.c* the skeleton of a complete C program with all the argument checking and basic declarations already built in. Copies of this file will be posted to net.sources or may be had by electronic mail request to either author.

People are always trying to build more complicated 'argv crackers'. To our mind, *getopt(3)* has proven satisfactory in the construction of dozens of small programs and the reshaping of dozens of others. *Getopt(3)* isn't IBM TSO's IKJPARSE, but then again, this is UNIX.

File */usr/pub/template.c* is too large to include here.

---

Scan command line arguments in a standard way

---

### 1.3.3. *cs*h - the self-contained shell

We will refrain from abusing the externals of *cs*h. The internals of *cs*h are every bit as bad, but lesser known.

*C*sh contains its own versions of *malloc* and *\_doprnt*. It is thus unable to use *stdio* or some other functions in the C library. When we attempted to link *cs*h with a faster version of *getpwent* that uses *stdio* and an *mdbm* data base for the password file, we found that *cs*h failed mysteriously.

Our fix was to make *cs*h use the old, slow *getpwent*. Such is the price of bad coding. (Neither of us uses *cs*h, so we don't care about its performance.)

Don't provide private versions of C library functions unless they are declared *static* (i.e. invisible to other object files).

---

Don't fight the C library; use it.

---

## 1.4. C Style and Portability

Coding style is in some ways a matter of personal preference. But there are some things that just don't work, or which are indicative of underlying sloppiness, or are just poor practice. Portability at a working level means the refusal to include code that is bound up in the shape of the particular digital computer we happen to be running on.

### 1.4.1. Magic numbers

Using a literal numeric constant in-line is almost always a bad idea, since the reader of such code is given little clue how the number was derived or what it represents. We return to 4.2BSD's infamous *ruserpass*.

```

ruserpass(host, aname, apass)
char *host, **aname, **apass;
{
    renv(host, aname, apass);
    if (*aname == 0 || *apass == 0)
        rnetrc(host, aname, apass);
    if (*aname == 0) {
        char *myname = getlogin();
        *aname = malloc(16);
        printf("Name (%s:%s): ", host, myname);
        fflush(stdout);
        if (read(2, *aname, 16) <= 0)
            ...
    }
}

```

Note the magic numbers: 0, 2 and 16. The first three zeroes are null pointers. The last zero is used to test for an error while reading the password or end of file (a zero count). The 2 is the file descriptor of the standard error output. 16 is the (arbitrary) maximum length of a password, including the null byte at the end.

```

#define MAXPWLEN 15
#define STDERR 2
ruserpass(host, aname, apass)
char *host, **aname, **apass;
{
    renv(host, aname, apass);
    if (*aname == NULL || *apass == NULL)
        rnetrc(host, aname, apass);
    if (*aname == NULL) {
        char *myname = getlogin();

        *aname = malloc(MAXPWLEN+1); /* 1 is for the null byte */
        printf("Name (%s:%s): ", host, myname);
        fflush(stdout);
        if (read(STDERR, *aname, MAXPWLEN) <= 0) /* error or EOF */
            ...
    }
}

```

These may seem trivial, but in larger programs it isn't always obvious that the many occurrences of some magic number are (or are not) related. `#define`'ing these numbers also makes changing them fairly easy. On the other hand, grabbing a random `#define`'d symbol that happens to have the right value, and using it, is no better. In particular, `stdio.h`'s `BUFSIZ` is often used incorrectly to mean "a bunch of characters".

---

Use `#define` to give numbers explanatory names.

---

#### 1.4.2. `_doprnt` considered unportable

Berkeley in 4.2BSD documented the internal `stdio` interface `_doprnt`. This was a mistake. Other `stdio` implementations often do not contain a `_doprnt` or anything like it. The Berkeley `curses` has used `_doprnt` for a long time, even before 4.2BSD:

```

/*
 * This routine actually executes the printf and adds it to the window
 *
 * This is really a modified version of "sprintf". As such,
 * it assumes that sprintf interfaces with the other printf functions
 * in a certain way. If this is not how your system works, you
 * will have to modify this routine to use the interface that your
 * "sprintf" uses.
 */
_sprintw(win, fmt, args)
WINDOW *win;
char *fmt;
int *args; {

    FILE junk;
    char buf[512];

    junk._flag = _IOWRT + _IOSTRG;
    junk._ptr = buf;
    junk._cnt = 32767;
    _doprnt(fmt, args, &junk);
    putc('\0', &junk);
    return waddstr(win, buf);
}

```

This is coded as if it were internal to *stdio*, yet it is not. The design error is in offering to provide an interface with a variable number of arguments, something which may be unimplementable on some machines and which on others is only expressed portably using `<varargs.h>`. A better design would have been to take a single string as an argument rather than a *printf* format and arguments. This would mean that the caller would have to format the arguments into a character buffer first using *sprintf* and pass the address of the buffer to this function.

No after version is offered,  
since the design is fatally flawed.

Don't assume that you can write functions that take a variable number of arguments. If you must do so, use `<varargs.h>`.

-----  
Avoid variable number of arguments in your functions.  
-----

### 1.4.3. Nested arguments vs checking

It's all too common to see a line of nested function calls. The main problem is that it discourages proper checking of system function return calls. This code, taken from *net.sources* in early 1985, is typical:

```
/* Blast into a users terminal. Great fun, and sometimes useful. */
```

```
#include <sgtty.h>
#include <stdio.h>
#include <sys/file.h>
```

```
int errno;
```

```
main(argc,argv)
```

```
int argc;
```

```
char *argv[];
```

```
{
```

```
    errno = 0;
```

```
    if (argc != 2) fprintf(stderr,"blast: need tty number (only).\n");
```

```
    else blast(open(argv[1],O_RDWR,0666));
```

```
}
```

```
blast(fd)
```

```
register FILE *fd;
```

```
{
```

```
    char c;
```

```
    if (errno) return;
```

```
    ioctl(fd,TIOCNXCL,0);          /* turn off exclusive use */
```

```
    while ( (c=getchar()) != EOF) ioctl(fd,TIOCSTI,&c);
```

```
}
```

Notice that the error message ('need tty number only') and the open call do not agree; the open seems to want /dev/ttyNN. But the open itself is not checked. Or is it? Notice the obscure line 'errno = 0'; the subprogram checks this and returns silently if errno != 0. I hate programs that exit silently on error conditions!

But that's not all. Note the sloppy declarations in the function. The variable 'fd' is returned by open(2), so it's an file descriptor (int). But it's declared in the subprogram as FILE \*, a **stdio** stream pointer. But it's *used* in the subprogram as a file descriptor again.

Here's how some of the code might look:

```

/*
 * blast - blast text into user's terminal input buffer (4.2 only)
 */

#include <sgtty.h>
#include <stdio.h>
#include <sys/file.h>

main(argc,argv)
int  argc;
char *argv[];
{
    int myfile;

    if (argc != 2) {
        fprintf(stderr,"usage: blast ttyname\n");
        exit(1);
    }
    if ((myfile = open(argv[1], O_RDWR, 0666) < 0) {
        perror("can't open terminal");
        exit(1);
    }
    else
        blast(myfile);
    close(myfile);
}

```

There are other problems in the subprogram; they are left as an exercise for the reader.

## 1.5. Coding

Coding errors and omissions are last, but not least, on our list of suggested improvements.

### 1.5.1. *isascii*, the forgotten macro

Programmers often seem unaware that most of the macros defined in `<ctype.h>` are only defined for arguments which are ASCII characters. 4.2BSD's *sendmail* sometimes uses these macros without first checking that the characters being tested are legal ASCII characters, via *isascii*. As a result, giving *sendmail* an address containing a character with the 0200 bit set will cause it to loop.

```

if (isspace(*s))
    s++;

```

This attempts to skip spaces, but will do undefined things if *\*s* isn't an ASCII character, possibly including referencing outside allocated memory, which may produce a core dump.

```

if (isascii(*s) && isspace(*s))
    s++;

```

This will stop skipping whitespace upon encountering a non-ASCII character.

-----  
 Use *isascii* before other *ctype.h* macros  
 -----

### 1.5.2. scanf sometimes stops scanning too soon

Programmers sometimes fail to test that `scanf` scanned as many items as they expected. This leaves the remaining variables pointed to by `scanf`'s arguments containing their previous contents, often trash if the variables are uninitialised. B news 2.10.1 contained such a bug in `rfuncs.c`, as shown below.

```
while (fgets(buf, sizeof buf, af)) {
    sscanf(buf, "%s %ld", n, &s);
    if (strcmp(n, ng) == 0) {
```

This code will leave garbage in `n` at end of file and in `s` at end of file or if the second item in `buf` isn't numeric.

```
while (fgets(buf, sizeof buf, af) != NULL)
    if (sscanf(buf, "%s %ld", n, &s) == 2 && strcmp(n, ng) == 0) {
```

This will only attempt to use `n` (and later `s`) if `sscanf` actually scanned two items.

-----  
Expect scanf to stop scanning inconveniently soon  
-----

## 2. "Code it now, we'll fix it later"

Careful coding takes longer.

'Later' never comes.

In a 'pressure-cooker' environment there is a strong tendency to 'get the thing out the door' without concern for software quality. I perceive this as a general failing of North American management; there is almost everywhere a pressure to provide the *appearance of productivity* regardless of true costs and long-range effectiveness.

The only answer to this is to fight bottom line with bottom line. If you add up the costs of one programmer-year for each major UNIX shop, to include the time spend porting 'portable' code, you will have a starting point for the real costs. Add in all the in-the-field debugging, including costs of debugging which are transferred to the end-user by shipping undebugged code, you'll be on your way. Don't forget to translate customer debugging time into customer dissatisfaction. My first guess at a conversion factor is

One unhappy customer == Ten lost sales

If you find your management pushing to you to 'code now, fix later', just remind them that 'later' never comes.

"Our competitors' code is done more carefully than mine.

I guess *their* bottom line extends to the horizon."

## 3. Conclusion

We can't close without citing two good guys. Despite its plethora of bugs and its `crypt(1)`-output-like configuration file, `sendmail` is careful about returning mail that would otherwise fall on the floor. The Honey Danber version of `uucp` is good in the same way, and may be better coded - we'll have to see.

A second, smaller-scale winner is the multi-key database (MDBM) posted to the net in mid-to late 1984. We built an entire password database structure on top of this package, and used it in a large student environment (several thousands of students over half a dozen UNIX systems) We ran across one obscure bug, but since the author, Chris Torek, had taken the trouble to check for "impossible" errors, we got the message "MDBM BUG..." instead of a scrambled eggs

database. Thanks.

Just to sum up, we've presented some guidelines for good programming and shown how they can reduce bugs. Reducing bugs means reducing costs.

Some of these guidelines are so well-known that they are almost truisms; many of these appear in your *fortune* file when you log in. Others are our own invention or are paraphrases of originals.

But guidelines are guidelines, and they are useful only if they are put to work in day-to-day programming. That's where you come in.

#### 4. Recommended Reading

For a view similar to ours, see Kernighan and Plaugher in *The Elements of Programming Style* (2nd Edition, McGraw-Hill, 1978) and the *Software Tools* books (by the same authors, Addison-Wesley).

For a countervailing view, see *net.sources*.

#### Acknowledgement

Thanks to Bruce Freeman for assisting with some of the examples. Our positions at the University of Toronto have afforded us the opportunity to peruse a tremendous amount of questionable code over the past 'N' years.

Careful reading by Laura Creighton eliminated a number of embarrassing errors from this paper.

Program fragments listed herein are copyright © by AT&T, The Regents of the University of California, and other interested parties. The *blast* program is by Mike Newton. The story of the UNIX and the Emveeous was inspired by Doug McIlroy's 'The UNIX and the Echo', of which our tale is but a pale reflection.

Some of the 'mottos' used are excerpted from *The Elements of Programming Style* (see under 'Recommended Reading').

Most of the paper was typed by one or another of the authors. Read 'we' for 'I', and 'I' for 'we', throughout.

# A Capability Based Protection Mechanism Under Unix

Daniel Klein

The Avatar Corporation  
5606 Northumberland  
Pittsburgh, PA 15217

(412)422-0285

{mcnc,decvax,floyd}!!dis!ml-cec!dvk  
Dan.Klein@CMU-CS-A.ARPA

## ABSTRACT

The Unix file system implementation is yet another example of a classical authority based protection scheme. Each file (or more correctly, each i-node) has associated with it a set of access modes based on the owner, group, and world. This protection mechanism is found in nearly all operating systems currently in existence. A different method of file protection is possible, and has been implemented in the Hydra operating system on C.mmp, and (to a large degree) in the hardware of the ill-fated Intel-432. This method is called a capability based protection scheme. In this protection strategy, file access may be given to users who do not necessarily fall into the arbitrary grouping schemes imposed by Unix. The access rights fall into more categories than a vanilla Unix provides, including read, write, append, and delete rights. Most importantly, the users obtaining access rights are determined by the owner of the file, and not by the system manager. This paper describes the implementation of a capability based protection scheme built on top of Unix. No modifications to the kernel are necessary, as all changes are implemented at user level. The capability protection scheme can be used with most standard Unix utilities *without* any modification.

## What Are Capabilities?

The Unix file system utilizes an authority based protection mechanism for restricting the access that individual users have on files, devices, etc. It uses a fairly common method of segregating file access rights into three classes, namely **read**, **write**, and **execute** privileges. Some operating systems also add **delete** rights, although Unix subsumes this into the **write** privilege on the directory containing the filename.

Unix also groups users into three general classes for determining which access privileges apply. Those classes are the **owner** of the file, the **group** to which the owner belongs, and everyone else. Some other systems (such as VMS) also include a **system** access class, although Unix shortcuts this distinction by allowing the superuser to do anything to any file. This presents some problems, namely that the

superuser can also easily delete things she did not really want to delete.

The important thing about the Unix file system is not so much *what* privileges are given out, but rather where they are stored. The key to the Unix file system is the inode, and all file access information is stored in the inode. This includes the owner and group fields, the protection fields, and the pointers to the file's data blocks. In essence, the file access restrictions are stored with the data object itself. The Unix file system has improved on most simple authority based systems by allowing multiple names (or links) for a single data object; nonetheless, by storing the protection restrictions with the data object, the Unix file system is a purely authority based system.

A capability based system, however, not only allows multiple names for a single data object, it also stores the protection restrictions with the name. Accessing a capability through a specific name gives the user a specific set of rights. In this way, accessing a data object via one name may afford the user greater or fewer privileges than if the user accessed the identical data object via a different name.

Now, as with the Unix file system, any user can create a data object and the capability to access it. However, capabilities can offer a greater degree of access checking than a simple authority based system. As an example, we can introduce "append" rights, which allow the user to access the data object only for appending, and not for arbitrary writing. (Berkeley Unix has introduced the notion of opening a file with an "append" flag set, but this does not prevent a user who is entitled to open the file for writing to do so without setting this flag. File access in Unix is done in a purely "gentlemanly" manner. File locks are merely advisory, and the only thing that prevents the superuser from reading your personal papers is the ethics of the administrator herself.)

However, a capability based system differs greatly from an authority based system in the manner of generating new names for existing data objects. In Unix, it is a simple matter — a new link is created using `ln`, and that is that. Because the access restrictions reside in the inode, I can easily make a link to a file which another user has read protected. However, I can no more access it with my new file name than I can with the original. Even if the owner of the file unlinks the original name, the protection fields (and owner and group information) remain intact, preventing me from circumventing the access restrictions.

In a purely capability based system, capabilities can only be copied from existing capabilities, and often this copying is restricted to the owner of the data object, or of the capability itself. When copied, capabilities can have privileges removed (but not added), allowing a user to give out a copy of a capability with "reduced power" to another user. That user can then potentially give away copies of her capability, but those copies are as restricted as the "original".

### What Are Capabilities Good For?

You are probably asking, "Okay, so what is the big deal?". The answer is best seen by pointing out the shortcomings of a standard

authority based system.

### Data Sharing

Let us say that I wish to share a file with another user, but with no one else. In Unix, the easiest way to do this is to create a new group, place both myself and the other user in the group, change the group owner of the file to the newly created group, and then set the file protections in such a way that group members can access the file. While this certainly works, it has a couple of big problems. First, as long as `/etc/group` is write protected (and certainly it should be), the creation of a "semi-private" file requires the intervention of a second party — namely the superuser. Second, the group file is going to get very big, very quickly, as users try to create these semi-private files. Third, your system administrator is very rapidly going to stop creating new groups as she gets annoyed by the sheer magnitude of requests. (One way of getting around the group access is to associate an "access list" with each file (this is the mechanism used by VAX VMS). This access list contains a list of users not in the proper group who may also access the file. The problem with this implementation is that there is a linear (or at best, a binary) search required to determine if a user is in the access list. To implement this mechanism in Unix would require a restructuring of the inode, and this is an undesirable approach. As can be seen in the following section, capabilities can be implemented in Unix without any changes to the kernel or file system.)

Because a capability is given away to a specific user (or set of users), only those users with capabilities for a data object will be able to access the data object. In this way, I can give access to a data object to a single user, and prevent all other users from accessing the data. Because I am not artificially restricted to the Unix "group" concept, I can share one file with one user, another file with another user, and a third file with a small set of users, as well as having "public" files. ("Public" files are created by placing a capability for a data object in a publicly accessible directory. Typically this capability will be a restricted one (for example, allowing only reading), while a powerful capability (allowing reading, writing, deleting, appending, etc.) would be kept in my private directory.) The concept of "groups" never enters into the picture, as *all* data access is done through the capabilities.

A simple example is one of proposal writing. If I am working on one proposal with Randy and Janet, I give them each a capability to that

proposal. I can give a capability to the second proposal to Hugh, and capabilities to a third proposal to Kim, Clem, and Veta. If at a later time I want to include Clem in the writing of proposal number two, I simply give him a capability to it. We do not have to be in the same groups, and we do not have to change groups to work on the proposals — we simply use the capabilities for the data. No other user can read or write these proposals, since they will not have the capabilities to do so. Similarly, Clem will not be able to read proposal number one unless I give him a capability for it.

### Distributed Access Privileges

Another concept which is easily implemented using capabilities, but which is only barely realisable using a standard authority based system such as Unix, is the notion of distributed access privileges. It is often very desirable (particularly in the implementation of data bases) to allow certain users to perform certain operations on the data, while allowing other users to perform other, potentially non-intersecting operations. For example, consider that you want some users to have the ability to read and write the data, or even to delete the file. Another set of users are allowed to read the data, or append to it. A third set would be allowed only to read the data. This much can *almost* be achieved with the current Unix file system implementation, (By protecting the file `-rw-rw-r--` and the directory that contains it `drwxr-xr-x` we come close, but not close enough. The owner of the file can read, write, and delete it. The members of the group can read and write it, but not delete it, and the rest of the world can only read it. This falls short of our goals, because we must rely on the group members' cooperation in only using the write privilege to append to the file. It also fails if the users we want to allow to read the file are fewer in number than "everyone else".) but it is impossible to implement if a fourth group of users is added who have the ability to append information, but not read the data (as in an audit trail), and a fifth group who may only read and write the file, but not delete it (as with a garbage collector).

The only way this mechanism can be realized is with capabilities. Each user in a certain set is given a capability to access the data in a prescribed manner. Since they must access the data through the capability, they can only do to the data what the capability allows them. The creator of the data starts with a fully unrestricted capability, and proceeds to distribute (restricted)

copies of it to the selected groups of users. Presumably, the users will maintain their capabilities in a private directory, so that there is no unwanted pirating of capabilities.

### Capabilities Under Unix

Capabilities are implemented under the Unix authority based file system in a very simple, straightforward manner. No changes to the kernel or the file system are necessary, and none are needed for most utilities. The implementation depends on four basic mechanisms inherent to Unix:

- 1) File access checking is done only on file open, not every time a read or write is performed.
- 2) If a file has been opened for reading, writing to the file is not allowed, and vice versa.
- 3) All open file descriptors (except those explicitly marked as `EXCLOSE`) are preserved across `exec` system calls.
- 4) The `setuid` feature on file execution can be used to allow a user to access files not ordinarily within her grasp.

A capability under Unix, therefore, is simply a small program. This program opens a file for some combination of reading, writing, and/or appending, and then `exec's` the program which desires access to the file. Because the file descriptor opened by the capability program is preserved across the `exec`, the called program can read, write and/or append through this file descriptor, accessing the data. Once opened, the file descriptor may be freely used to manipulate the data file, so long as these manipulations fall within the restrictions established by the `open` system call.

Now the data which is to be accessed is undoubtedly protected against normal Unix file access. The capability program therefore must be `setuid` to the owner of the data. For the duration of the capability program execution, all file access checks are done as if the user accessing the data were the owner of the capability. If the owner of the capability and the owner of the data are the same, then the person using the capability can access the data in the same way as the owner of the data. But because it is a program that is doing the access, the capability will only grant access to the data that its creator specified — and

not any file the user desires. Once the file has been opened, the capability resets the effective userid of the person running it back to the real userid, effectively preventing any further special access privileges for this process (unless another setuid program is run).

The only catch is telling the called program which file descriptor to use — but this is easily accomplished by passing the descriptor number as a parameter to the program. Another mechanism would be to have the called program do an *fstat* on its open file descriptors. In any event, once the capability *exec*'s the user's program, that program has an open file descriptor with which to access the desired data.

### An Implementation Example

Let us now look at an example of a capability program, and its use. The following program is all that is required for a capability. The only differences between capabilities are:

- 1) The actual data to which they refer. Since we are still operating under Unix, the data has a Unix file name, referenced by the variable *capfile*.
- 2) The access modes allowed by this capability. They are expressed by the flags passed into the *open* system call, and are stored in the variable *how*. (For the newer 4.2 BSD *open* system call, we would also have to store a second creation mode field).
- 3) The ability to make copies of this capability. (An actual *copy* of a capability is useless, since it will not be setuid to the proper person. Only a link (an alternate name) for a capability will have the desired results.) This is encoded into the variable *who*. This restriction looks like an access list (implying a database lookup), but instead it simply checks to see if the user of the capability is the owner of the capability. A zero value means that anyone may use (and copy, by making another link to) the capability, while a non-zero value means that only the intended recipient may use the capability.

Because only a small difference exists between individual capabilities, a simple automated program (*mkcapa*, for example) could be used to generate capabilities given a filename

to reference, an access mode, and an intended user. This program could then directly produce an executable file (no need to recompile it every time) which was setuid to the creator of the capability.

Here is the C program associated with a capability which will be referred to as *getit* in the following examples. *Getit* is an arbitrary choice of names. It is up to the user to find some mnemonic correspondence between the name of the capability and its function.

```
#include <stdio.h>
/* This is information specific to a capability */
char *capfile = "/usr/whoever/file";
int how = 2;
int who = 143;

/* This is code common to all capabilities */
main(argc, argv)
char **argv;
{
    int ifd;
    int passit = 1;
    char sfd[10];
    char *execname;
    char *execargs[128];
    char **execv;

    if (who && who != getuid()) {
        fprintf(stderr, "Illegal userid for %s\n",
                *argv);
        exit(1);
    }
    argv++; /* Skip argv[0] */
    if (strcmp(*argv, "-") == 0) {
        close(0); /* Wants to use stdin */
        passit = 0; /* Don't pass on ifd */
        argv++; /* Skip over the flag */
    }
    else if (strcmp(*argv, "+") == 0) {
        close(1); /* Wants to use stdout */
        passit = 0; /* Don't pass on ifd */
        argv++; /* Skip over the flag */
    }
    ifd = open(capfile, how); /* Open the file */
    if (ifd == -1) {
        fprintf(stderr, "Capa access failure on %s\n",
                capfile);
        exit(2);
    }
    execname = *argv++; /* Get name of exec file */
    for (execv = execargs; *argv; argv++, execv++)
        *execv = *argv; /* Copy argv to execargs */
    sprintf(sfd, "%d", ifd); /* Convert descriptor number */
    if (passit) /* And if indicated, */
        *execv++ = sfd; /* Write to the arg list */
}
```

```

*execv = 0;          /* Terminate the list */
setuid(getuid());     /* Change to real userid */
execv(execname, execargs); /* away we go... */
exit(3);             /* Just in case */
}

```

## Using Capabilities Under Unix

Obviously, the program can stand some refinement. We have not considered the search path, or environment variables — but this is just to serve as a simple example.

Let us now examine a few sample cases that use capabilities.

### Specialized Programs

To use the capability defined above (let us assume it is called **getit**) with the program **hackit** (with a few flags), we would type the command:

```
getit hackit -x -y
```

The program **hackit** would then be invoked with the arguments:

```
hackit -x -y 3
```

where “3” is the descriptor number that the file accessed by the capability is open on. Now all that **hackit** needs to do is:

```
capafd = atoi(argv[argc-1]);
```

and the file associated with the descriptor number in *capafd* can be read as if **hackit** had opened the file itself.

In fact, because a capability simply opens the file it is directed to, and then executes the remainder of its argument string, multiple capabilities can be invoked for a single program. If we have two capabilities, called **inputter** and **outputter**, and a program that needs to read one file and write the other (both accessed by capabilities), then the command:

```
inputter outputter prog
```

will first run the program **inputter** (which will open the first file, tack the file descriptor number at the tail of the argument list, and *exec* the remaining arguments). This will cause the following command to be executed:

```
outputter prog 3
```

This will cause the program **outputter** to be run (which will open its file, append its file descriptor number at the tail of the argument list, and *exec* the remaining arguments. So **prog** will finally be run with the arguments “3” and “4”, which are the file descriptor numbers on which it can read and write the appropriate data files.

## Programs That Read *stdin* And Write *stdout*

The capability mechanism diagrammed above allows the user to dynamically specify that the capability should open the data file on *stdin* or *stdout*, instead of the next available descriptor. Because of this, the user can use not only specialized programs, but most any of the Unix utilities. For example, if the user wished simply to **cat** the contents of the data file referenced by the capability **getit**, the command:

```
getit - cat
```

would invoke **cat** with its *stdin* file descriptor redirected to the file accessed by the capability. If the user wished to redirect the output of **ls** to the file, the following commands would do the trick:

```
getit + ls
```

-or-

```
ls | getit + cat
```

In the first command, the standard output of **ls** would be redirected to the file accessed by the capability. In the second, **cat** would have its standard input redirected to the pipe by the shell, and its standard output redirected to the file pointed to by the capability.

In a similar manner, an ordinary Unix utility is to be used to copy some portion of one file (accessed by a capability) to another file (also accessed by a capability). If the **grep** utility was to be used to find all lines with the string “Klone” in the file accessed by **inputter**, and write those lines to the file accessed by **outputter**, then the command:

```
inputter - outputter + grep Klone
```

-or-

```
outputter + inputter - grep Klone
```

would be all that was needed. Since **grep** ordinarily reads *stdin* and writes to *stdout*, the previous command is equivalent to:

```
grep Klone < inputter > outputter
```

assuming, of course, that **inputter** and **outputter** were ordinary files instead of capabilities.

### Using vi

Capabilities would be useless if they did not allow users to run an ordinary editor on the data that they reference. Since **vi** and **ed** do not read *stdin* to get the data to manipulate, we run into a slight problem. Of course, the following sequence of commands could be used to circumvent the problem:

```
capa - cat > temporary
```

```
vi temporary
```

```
capa + cat < temporary
```

```
rm temporary
```

This will work, but it means that the file

**temporary** is vulnerable to copying, even though the capability **capa** may have been designed to prevent this.

There is a better way. **Ed** is a lost cause (in more ways than one, thank you very much), but **vi** thankfully has a feature which makes it usable with capabilities. The "filter" command allows the user to pass the **vi** data buffer through a program (or programs), and replace the buffer with the output of that program. Thus the command "**1G!Gnroff**" will go to the first line of the file (**1G**), and filter the entire file (**!G**) through **nroff**.

In a similar manner, the user can enter **vi** (with a null file name) and type the command "**1G!Gcapa - cat**". This will run **cat** as a filter, with its input being the data file referenced by **capa**, and its output being the **vi** buffer. Once the data file has been changed, it can be written out with the command "**1G!Gcapa + cat**". The input to **cat** will be the **vi** data buffer, and the output will be the file referenced by **capa**. Best of all, no temporary files need be created (except those that **vi** ordinarily creates), so the security of the data file referenced by **capa** will not be compromised.

### Additional Features

Because capabilities are added to an existing file system, certain additional features can be added in to those not normally found in fully capability based operating systems.

### Security And Naming Issues

Obviously, some naming convention will need to be established to associate the name of a capability with the name of the file that it references, although this is not strictly necessary. The user *need not know* the name of the file she is accessing — only the name of the capability. If the capability program is not readable, then the name of the referenced file can be kept private, adding a further level of security to the Unix file system. If the referenced file is a symbolic link (which is unreadable by users other than the owner of the capability), then even the prying eyes of **fstat** and **lstat** can be kept shrouded.

### Capability Control

In a normal capability based system, a capability serves a similar function to a link. Once a capability has been given away, it is a difficult task to find all the copies. The **find** and

**ncheck** utilities are used on Unix to locate all files with a specific inode number, but this requires searching some large subset of the file tree. This is also the case in a fully capability based system. If the user wishes to find out where capabilities have been copied to (possibly for control of who has access to them), then a very I/O intensive process must be undertaken.

With capabilities under Unix, this is also the case, but Unix has one great advantage in its favor. If I, as the creator of a capability, wish to invalidate a capability (for example, I decide to remove a user's access to a file), then all I need do is copy **/dev/null** into my link to his capability. Since each "copy" of a given capability is really just a link to it, zeroing out the capability invalidates all "copies" of it. However, the data file remains intact, and any other capabilities that referred to the same data are likewise still valid.

In this way, a user can not only give out specific access rights to selected individuals, she can also take back those rights, or reassign others. A truly flexible file protection mechanism can thus be realized. This also implies that there will be many capability files for each data file, and that this can result in a higher overhead. The overhead is small, though, and is something one expects to pay for increased flexibility.

### Capabilities and Sockets

Berkeley 4.2 Unix allows users to pass file descriptors across a connected socket through the **sendmsg** and **recvmsg** system calls. Because of this, the capabilities that are to be used by a program do not have to be determined before the program is run. Instead, the name of a capability can be given to a running program, and the following sequence of actions can be used to dynamically pass capability accessed file descriptors. As with everything else in 4.2, the capability program will have to be modified slightly.

- 1) The program wishing to access the capability maintained data performs a **socket-pair** operation followed by a **vfork**, creating two processes linked together by sockets.
- 2) The child process does an **exec** on the capability program.
- 3) The capability program (child process) **opens** the file, and does a **sendmsg** to its socket, transmitting the file descriptor of the data file to the parent process.

- 4) The parent process does a *recvmsg*, accepting the file descriptor. The parent now has a file descriptor that refers to the data file accessed by the capability.
- 5) The child process exits, and the parent continues (reading or writing on the file descriptor given to it by the child).

A different first argument to the capability program (for example “++”) would indicate the socket transfer of the file descriptor. A utility routine (called *opencapa*) would do all the hard work for the user. All the user would pass in would be a capability name. The routine would return a file descriptor number, or an error indication.

### Examples Of Capability Usage

I would like to present some real-life examples of the use of capabilities. Understandably, the examples are simple, although they should give the reader an idea of the power of a capability based protection mechanism.

#### Bulletin Board System

Consider a system with multiple bulletin boards. Some bulletin boards are readable by everyone, but other bulletin boards are to be read only by certain people. To make things difficult, the groups of people can partially intersect each other. Each bulletin has its own moderator (or moderators), and one person is in charge of the entire system — creating new bulletin boards or deleting old ones as necessary.

To implement this, the bulletin board files are all owned by the person in charge. This person gives out read/write capabilities to specific bulletin boards to the moderator(s). If necessary (i.e. a non-public bulletin board), individual users can be given capabilities to read the board. A capability to append to the boards is placed in a public (or private) place to allow users to post to the boards.

There need only be one general program to read the bulletin boards, one to post to them, and one to purge old or unwanted postings. Each of these programs is called with the appropriate capability. In this way, the creation of a new bulletin board does not mandate the changing of the posting program. The posting program does not even know about the filenames involved in the bulletin board system — all it sees is a file descriptor to manipulate. The same is true for

the reading and moderating programs.

#### A Payroll System

In a payroll system, multiple levels of protection are needed. A single file is used to store the payroll information, yet multiple users will access it. The treasurer, president and head of personnel are given capabilities to read and write the data file, so that they may examine and change salaries, as well as process employee terminations. The cashiers are given capabilities to read the data file so that they may disburse salary checks, but they may not alter the data in the file. The remaining members of the personnel department are given capabilities to append the data file (but not read, and not arbitrarily write) for processing new employee hirings. No one else may access the file in any way.

No convolution of groups will realize this distribution of protection. Capabilities, on the other hand, easily solve this problem. The actual data base is kept in some secure location, protected “read/write” only by the owner (i.e. the treasurer). The president and head of personnel are each given a read/write capability, while the cashiers are each given a read-only capability. These may each be links to the same capability, or individual capabilities. In the latter case, the capability of one cashier can be invalidated (leaving the others intact) when that person transfers to another department. Finally, the remaining personnel department members are given append capabilities.

Because the data file itself is protected against reading and writing, the users on the system (whether they have capabilities to the data or not) cannot arbitrarily read or write the data file. Only through using their own capabilities can they access the data, and then only in the prescribed manner.

### Conclusions

Capabilities allow for a much more generalized file access method than is realisable in standard authority based systems (such as Unix). Capabilities can be added to Unix without any changes to the kernel, file system, and most utilities. The mechanism is sufficiently simple that it can be incorporated into V6, V7, System III, System V, and Berkeley 4.2 Unix implementations. The inclusion of capabilities greatly enhances the functionality of the Unix operating environment without incurring an unacceptable expense of implementing them in the kernel. Because

capability based file protection distributes data access restrictions, problems which would otherwise be inadequately or incompletely implementable are now easily solved under Unix.

### Acknowledgements and a Request

The author wishes to publicly thank Hank Mashburn, Bill Wulf, Joe Newcomer, and the rest of the CMU C.mmp Hydra project for turning me on to these concepts back in 1976. Capabilities continue to amaze me with their flexibility, functionality and applicability.

I would also like to request that anyone who uses this mechanism in a project or product be kind enough to let me know about it. To design is human; to actually implement, divine.

# An Overview of the *SETOPT* Command Line Option Parser Generator

Gary Perlman

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

*SETOPT* is a set of macros for generating C functions to parse command line options for UNIX System (TM Bell Laboratories) commands. A simple language describes all the options allowed with a program. Each option is defined by its name, purpose, data type, size, range, and other information. This information helps generate program code to standardize the interface between users and the program, and between the programmer and the parser. Programmers can assume a perfect user interacting with their programs because on-line help and diagnostic messages tell users what is available and makes sure that legal option values are supplied. Extra abilities include generating summary documents and menu/form-filling interfaces.

## The Problem

With no consistent syntax to UNIX System commands, people are bound to have problems learning and using commands. UNIX System commands consist of a program name, a set of options that control the behavior of the program, and arguments (usually file names). Traditionally, these options are single characters preceded by a dash, and if a value is to be supplied (logical true/false options take no values), it follows the option. For example, a simple text formatter, *tf*, might have an option to delete leading blank space on lines (-d), and one to control the width of lines (-w), and be called with options as follow.

```
tf -d -w 78
```

This syntax is a simplification because some options for UNIX System commands are multi-character names, and some are not preceded by a dash. Some values for options immediately follow the option name, while others are preceded by a space.

Hemenway and Armitage (1984) proposed a syntax standard based on an analysis of the standard set of over 400 UNIX System commands. Their proposal included a set of 13 rules for the syntax of commands and options and was meant to insure the consistency of commands developed in the future, and to guide the gradual migration of old commands toward a standard. Software to

aid programmers in following the standard was recommended.

The published aid to parsing command line options, *getopt*, has some differences with the Hemenway and Armitage (1984) standard, but with some minor modifications, it could be consistent with the proposed syntax. To use *getopt*, programmers supply the names of the options and the ones taking values. For example, the string "abx:y:" defines two logical options, a and b, and two taking values, x and y. This *getopt* string is passed as an argument to the *getopt* function along with the unprocessed command line options, and *getopt* returns each option and its value (if appropriate) one by one. *getopt*'s sole purpose is to help parse the command line options.

The user interface provided by programs using *getopt* depends on programmer efforts. No standard on-line help nor diagnostic messages are provided; programmers must write their own. Programmers are also required to write code to check the data type and range of values supplied, and do any conversions and assignments of the basic string-type option values to internal variables of type integer, real, and so on. Many programmers do not do a good job at these tasks, leaving values unchecked or providing inconsistent error messages. Most do not do them all.

## The Solution

*SETOPT* is a collection of text generators written in the macro text processing language *m4* (Kernighan, 1983). *SETOPT* helps programmers deal with command line options. A simple language describes the attributes of each option for a command, and this specification is expanded into C language functions to automate most of the tasks involved in dealing with options. The user interface provided by *SETOPT* is one that allows users to request help with options, gives diagnostic error messages when supplied values are incorrect, and promotes a consistent syntax that users have to learn once. The interface to the programmer allows the programmer to call the generated parser once, and when it returns, an error is reported, or the options are known to have been validated and set.

The language describing command line options is based on the interface language used by the S system for data analysis (Becker and Chambers, 1984). The S system uses a command language in which each option to a function has a name, a data type (like integer, or character), a size, a default value, whether it is required or not, and some other information. While the syntax for UNIX System command line options is different, and the requirements for the options differ, the S system interface language makes sure that options are correct, and that is the most basic goal of an option parser. *SETOPT* is more advanced than the S interface system in that it allows more user interaction and on-line help.

The *SETOPT* option specification language is based on two macros:

```
PGM(NAME, HELP, PROCESS)
OPT(FLAG, VAR, HELP, TYPE, SIZE, DEFAULT,
    TEST, ACTION)
```

written using the general purpose *m4* macro pre-processor (Kernighan, 1983). The PGM macro specifies the name and purpose of the program, and the data type of arguments processed by the program; these are used in on-line help and error messages. The OPT macro specifies the attributes of UNIX System command options for the program.

**FLAG** A single character name for the option to be used by the program user. This is checked by *SETOPT* to make sure it conforms to the standard of being alphanumeric.

**VAR** A program variable that will be set

by *SETOPT* to be used by the programmer. This is used to name a component in a data structure that contains all the options.

**HELP** A short phrase describing the option.

**TYPE** The data type of the option. It can be one of: logical, string, character, integer, real, file, directory, date, time.

**SIZE** A size of zero implies a scalar value while larger values imply an array of values.

**DEFAULT** The default value of a scalar option if the user does not specify it.

**TEST** An expression to test the validity of a supplied option value.

**ACTION** Actions implied by setting an option. For example, setting one option often implies other are set.

These option description parameters evolved with use of *SETOPT*. For example, new data types were added as their need became known, and others might be added. Also, some parameters have been added (e.g., implied actions) after it was realized that their inclusion would reduce programmer effort. Other parameters that might be supported in the future include a way to require certain options be set and allow pre-validation transformations (e.g., mapping to upper/lower case to allow for more general date formats).

The programmer describes all allowed options and runs a *SETOPT* program to generate a function to parse those options. Generating a parser has advantages over having a single parsing function to handle all possibilities. The generated parser is as compact as possible to deal with the data types and ranges of options particular to a given program. A parser that could check all kinds of data types and test range validity with a variety of expressions would be large and adversely affect portability to small systems. With a generated parser, if specific types of options, say dates or times, are not used, then code to test for them need not be generated. Generated code uses compiled validity test expressions, which are faster than any interpreted function. This is not a large saving because command line option parsing is done once, and for a few options. The real saving is the space that an interpretive expression evaluator would take.

When the programmer calls the parser, usually the first statement in a program, several activities take place before it returns.

- \* Each option and any value are taken from the command line.
- \* The option is checked to make sure it is recognized. If not, an error message is printed.
- \* If it takes a value, *SETOPT* checks to make sure there is a value supplied. An error message indicates when an option value is missing.
- \* The type of the value is checked. A message indicating the correct type is printed for invalid inputs.
- \* The option, in the string format typed by the program user, is converted to the correct type format and assigned to the program variable named in the option specification language. This variable is an entry in a structure defined in a header file created by a *SETOPT* generator. Access to this structure is done with macros to simplify the programmer's task and guard against possible changes in the structure standards.
- \* The validity (for example, range) test is applied to the converted value. If there is an error, the value is reset to the default value after an error message is printed.
- \* Any implied actions, like assignments, are made.

If an error is detected, a standard error message is printed naming the program and the problem. Inappropriate actions are not taken after an error; for example, implied actions take place if all previous activities were error free. When errors are detected, a negative value is returned by the parser. Usually, the program will exit after an uncorrected error. Note the detail required to deal properly with just one option, and the need for a programmer's attention to make sure that error messages conform to standards. Besides making the parsing task easier for programmers, *SETOPT* makes sure that option handling is high quality.

*SETOPT* provides several built-in options available in every program. Some of these are hand programmed into many programs.

help	Get on-line help about the attributes of all the options.
synopsis	Get a short synopsis of the command usage.
version	Get the version date of the program

options.

set	Set options interactively. This combines with on-line help to provide a menu interface.
read	Read options from a file.
done	Signal the end of options. This is part of the syntax standard and allows program arguments (like files) to begin with a dash.
exit	Exit from the program.
shell	Temporarily leave the program to run another program.
chdir	Change the working directory of the program.

Because any effort put into *SETOPT* affects all the programs using it, the large programming task of adding many built-in options is economical.

### Additional Features

*SETOPT* is designed to allow simplified versions for aesthetic, security, or program-size reasons. The generated parser has program code for all the operations described in this paper, but not all have to be compiled into the final program. Along with the parsing program code are special control statements (for the C compiler pre-processor) so that the programmer can independently control at compilation time whether on-line help is to be supplied, options can be set interactively or from files, or option values are checked.

*SETOPT* can help document programs by generating summaries of options based on the option specification language. Even programmers who take the time to write documentation find it is difficult to keep the documentation up to date with the program code, and often, important details are omitted. When the option parser and the documentation are generated from the same source, as in *SETOPT*, the documentation is guaranteed to be accurate. For the parser, *SETOPT* generates C program code, while for the documentation, *SETOPT* generates text in the *troff* text formatting language used on the UNIX System. Part of an example manual entry is shown at the end of the paper. A collection of target languages, including a user interface programming language (Vo, 1984) that provides a menu and form-filling interface to programs, are possible and have been prototyped.

## Conclusion

*SETOPT* provides programmers with a tool for handling command line options for UNIX System commands. People using *SETOPT*-based programs benefit from a consistent user interface with standard built-in option and standard error messages. To the extent that the same conventions are used in a large set of commands, people will find programs easier to learn and use; there is activity to standardize the user interface of all UNIX System commands. From the programming perspective, programmers are spared the tedious programming of the routine chores of parsing and checking options and providing on-line help and error messages. They do not have to worry about meeting any syntax standard because *SETOPT* generates parsers that follow the Hemenway and Armitage (1984) standard. In addition, programmers are able to generate up-to-date user documentation with minimal effort.

## An Example

The example below shows part of the option specification for a simple text formatter. From this compact description, about 400 lines of parsing code are produced, resulting in a substantial time savings for programmers. With no extra effort, all the information is available to users in a more humane format, both on-line and in standard UNIX System manual format. Removed from the example is supplementary text, added by the programmer, to make the manual entry more informative. Some of this is shown in the sample manual entry at the end of the paper.

```
PGM(tf, Simple Text Formatter, FILES)
OPT(b, breaklines, Break ALL Lines of Text)
OPT(c, center, Center Input Lines, LGL,
    0, FALSE)
OPT(d, delspace, Delete Space Around Lines)
OPT(i, indent, Indent Output Lines, INT,
    0, 0, value >= 0)
OPT(j, justify, Justify the Right Margin)
OPT(N, number, Number Output Lines, LGL,
    0, FALSE)
OPT(p, paginate, Paginate Output)
OPT(s, spacing, Output Line Spacing, INT,
    0, 1, value > 0)
OPT(t, tabs, Absolute/Relative Tab Stops,
    INT, 20)
OPT(w, width, Width of Output Lines, INT,
    0, 72, value > 0)
```

## Availability

*SETOPT* was developed at AT&T Bell Laboratories and they retain the rights to its distribution. Their plans for *SETOPT* are not known. Since leaving Bell Labs, I have been developing a new version of *SETOPT* based on the details in this publicly released paper. If AT&T does not force the issue, I will be happy to distribute the macros for the parser and manual entry generators. These macros depend on the version of *m4* distributed with UNIX System V, although a version may be possible without the dependence on its new features.

In the meantime, I recommend using the simple *getopt* command line option parser. It will help standardize the user interface of your programs and it will help structure your code so that migration toward the Hemenway & Armitage syntax standard or toward an advanced parser like *SETOPT* is simplified. Although part of System V, there are public domain versions of *getopt* that have been posted to the UNIX System network.

## Change of Address

More information about *SETOPT* can be obtained from me at my new address:

Professor Gary Perlman  
Wang Institute of Graduate Studies  
Tyng Road  
Tyngsboro, MA 01879  
(617) 649-9731  
wivax!perlman or sdcs!perlman

## References

- Becker, R. A. & Chambers, J. M. (1984) *S: A Language and System for Data Analysis*. New York: Wadsworth.
- Kernighan, B. W. (1983) *m4: A Macro Preprocessor*. Standard UNIX System documentation.
- Hemenway, K. & Armitage, H. (1984) Proposed Syntax Standard for UNIX System Commands. Paper presented at the 1984 Winter USENIX conference, Washington, D.C.
- Vo, K.-P. (1984) Integration, Interaction: The IFS Approach. Paper submitted to the AT&T Bell Laboratories Technical Journal.

## NAME

tf - Simple Text Formatter

## SYNOPSIS

tf [-bcdjnp] [-i indent] [-s spacing] [-t tabs] [-w width] [-] [files]

## SUMMARY

Option	Purpose	Default	Range
-b	Break ALL Lines of Text	FALSE	
-c	Center Input Lines	FALSE	
-d	Delete Space Around Lines	FALSE	
-i I	Indent Output Lines	0	value $\geq 0$
-j	Justify the Right Margin	FALSE	
-n	Number Output Lines	FALSE	
-p	Paginate Output	FALSE	
-s I	Output Line Spacing	1	value $> 0$
-t I20	Set Absolute/Relative Tab Stops		
-w I	Width of Output Lines	72	value $> 0$

## DESCRIPTION

tf is a simple text formatter. It formats plain text files by filling, spacing, indenting, numbering, centering, or setting tabs based on program options. This program has *all the options*, almost.

tf processes the named files, but if no files are specified, the standard input is read. If a single dash is supplied as an argument, the standard input is read there; this allows inserting the standard input into a list of files.

## OPTIONS

The following options are allowed with tf. Options should be flagged with a dash (-) followed by the option character. If the flag takes a value, then it should be the next string on the command line. In the options below, option characters followed by a letter take values. I flags are followed by an integer. Logical flags (TRUE/FALSE variables) take no values.

tf begins by reading options in a file in your login directory called .setopt/tf (if it exists). Each line of this file should begin with an option character followed by a value, if appropriate. Do *not* use the flag option indicator (-). After reading the option file, options on the command line are processed.

## Detailed Description of Options

<detail omitted>

# Geritol for Old Programs or Troff's Got a Lot of Life In It Yet!

Robert P. Lawson, SoftQuad Inc.

Avi Nalman, SoftQuad Inc. and University of Toronto

David Slocombe, SoftQuad Inc.

Mathew Zaleski, SoftQuad Inc.

Box 744, Station P,  
Toronto, Ontario M5S 2Z1  
(416) 593-4275

## ABSTRACT

Troff (ditroff) is an old program, but not so old that it cannot be enhanced and given a new lease on life. We have added a hyphenation exception dictionary, character-kerning, a logical-to-physical font mapping, font-table caching, a new, more compact and more informative intermediate language, long fontnames and long macronames, and we have improved the maintainability and speed of the program. As a result, troff is now able to drive a greater range of devices, produces higher-quality type in general, and will (we hope) refrain henceforth from eating VAXen for breakfast.

## Introduction

Troff -- in Documenter's Workbench, 'troff' means 'ditroff', while the old C/A/T-specific troff has been renamed 'otroff' -- can be a flexible tool for formatting text on phototypesetters and laser printers, and a truly spectacular performer when used with dot-matrix printers instead of the word-processing packages usually employed.

However, before troff can fulfill its promise, many problems must be overcome:

(1) Troff does not perform the fundamental H&J (hyphenation and justification) task in a manner that meets the high standards of commercial typography. For example, troff has no hyphenation dictionary, exception or otherwise. This is, perhaps, not a major crime in an academic environment since troff's very conservative approach to choosing hyphen-points ('If you aren't sure, don't try it.') results in relatively few bad hyphenations. But the resulting large wordspaces are ugly and slow the reader down. More significant to the commercial book-manufacturer, inadequate hyphenation can add whole signatures to the length of a book, thus

increasing costs.

(2) Character-kerning is also lacking in troff. The character-pair 'To' is a representative example of the need for kerning: with proportionally-spaced fonts the 'o' should be tucked closer to the 'T' than normal width would have it. Without kerning, minimum wordspaces have to be larger than desirable because words are not guaranteed to 'hang together' properly. This in turn requires a larger-than-desirable leading (line-to-line spacing). It all adds up to a bigger paper or book.

(3) Troff is not truly device-independent. In too many ways it still assumes that its output device is a photomechanical composing machine such as was universal in the 1960s and 1970s. Many characteristics of modern digital typesetters and laser printers -- not to mention the more flexible of the dot-matrix printers -- simply do not fit well with troff's world-model.

(4) Troff is slow. Indeed, troff is well known to eat VAXen for breakfast, and people with small UNIX boxes driving dot-matrix printers may find themselves eaten for dessert. There is a finite supply of VAXen, and sooner or

later troff will eat the last one and the world will end, unless something is done about it.

EXCEPT...

Do not meddle in the affairs of troff,  
for it is subtle and quick to anger.

--utzoo!henry

With these wise words ringing in our ears we have donned our armour and tackled troff head-on . . .

### Better Hyphenation with an Exception Dictionary

We decided not to mess with Ossanna's hyphenation algorithm itself -- at least not just yet. Instead we forced troff to check for entries in a disk-based table of words (or word-roots) known to be incorrectly (or inadequately) hyphenated by the algorithm.

In this we were motivated by the belief that any improvement in the algorithm itself was unlikely to make an exception dictionary redundant, so we might as well get its creation over with.

The algorithm Ossanna used was a variant of the 'Time Magazine' algorithm, which first strips off suffixes (assigning hyphens where possible) and then guesses the probability that hyphens are allowed between pairs of characters in the remaining word-root. Another variant of this procedure was known to work adequately in a newspaper typesetting program written just prior to troff's creation, provided it was coupled with an exception dictionary of roughly 5,000 words and word-roots.

While there is hope that TEX might provide us with a better algorithm [Liang, 1983], we could not convince ourselves that it was likely to make an exception dictionary unnecessary.

The chief requirement in constructing the dictionary lookup mechanism is to keep words with common roots together in one block as much as possible. This is because the dictionary can be kept much smaller if, wherever possible, only word-roots are used; then the one word-root corrects the hyphenation of a host of words with the same root but different suffixes or suffix-strings. For this to work, it is necessary to check the dictionary after stripping each suffix from the candidate-word. This is no problem if a word

and all the stripped word-roots (e.g., 'dictionary', 'diction', and 'dic') would each occur in the same disk-block if it were in the dictionary at all.

The problem which then presented itself was how to keep the random disk accesses associated with the hyphenation exception dictionary from eating up a significant portion of the execution time which we hoped to save in other ways. There can be a very large number of attempts to hyphenate, approaching once per line when text is set in narrow columns. If each such attempt caused a disk access, then the ultimate speed of troff would be bounded by these disk operations alone.

Fortunately, a 'Bloom filter' offers a way around this problem ([Bloom, 1970], [Gremillion, 1982]). The idea is to use a bitmap of 2 KB or so: for each word in the exception dictionary you turn on the bits indexed by several hash functions of the word. Then, before going to the disk to check on the existence of an exception word, you can hash the candidate word in an identical manner. If each of the bits indexed by each of the hash-functions of this word is on, then the candidate-word may (possibly) be in the dictionary. However, if one or more of the indexed bits is not on, then the word is certainly not in the dictionary and there is no need to access the disk to check on it.

Although the Bloom filter does not eliminate all unnecessary disk accesses, it can come close enough to make the exception dictionary no threat to our speed-up efforts.

### Fixing the Character-fit with Kerning

In the five-hundred years of printing, kerning has been more talked about than done.

With hand-set type it was possible to segregate some copies of particular letters and file down the sides of these letters so that, in combination, they would fit better. But it was practical to adjust letter-fit this way for only a few of the character pairs that could benefit from better spacing.

With the invention (and virtual dominance in America) of the Linotype machine, which cast a lead 'slug' for a whole line at once, kerning became something most typographic craftsmen just dreamed of. The early photomechanical typesetting machines were based on very primitive electronics, and, when computers began to replace the 'hardwired logic', their memories were

so limited that the kerning tables which appeared in these machines covered little more than the cases which the old craftsmen had managed to handle with their specially-filed-down types.

Today memory is less of a concern. Yet a 100-character font would need a 10,000-cell kerning table if all character-pairs were to be considered for adjustment. Apart from the problem of creating these 10,000 numbers (many of which, admittedly, would be zero), there is the fact that a typesetting house may have upwards of 1,000 fonts in its font-library. And kerning across fonts (say, an italic 'f' with a roman 'a') is out of the question with kerning-pair tables.

So we have used a more recent approach, called 'sector kerning'. Each letter has, stored with its basic width, a set of 'penetration values' -- 3 to 5 numbers -- for its left side and its right side. These numbers represent the distance from the edge of the box enclosing the letter to the edge of the letter itself, within one of the horizontal bands ('sectors') into which the box is partitioned. In a crude way, the penetration values represent the shape of the letter.

For any pair of letters (subject to limitations when the letters are not from the same font), the right-penetration values of the left character can be added to the corresponding left-penetration values of the right character and the minimum of the resulting sums can be computed. This minimum is the key determining factor in the amount of kerning to be used.

It turns out that the amount of computation involved in calculating a kerning value from the penetration values 'on the fly' is not so bad compared to accessing kerning-pair tables, which must involve some trick or other to compress their sparse structure.

We expect that the kerning mechanism which we have implemented will have an especially beneficial effect on the appearance of the poorer font-designs which are found on cheap dot-matrix and laser printers. The creators of these fonts were often the programmers of the firmware, who did not realize that there was even a problem making letters fit well together, while the most famous traditional fonts used in typesetting have lasted decades or centuries precisely because their letters fit reasonably well together even without kerning. Not that these better fonts won't benefit from kerning: but only experts are likely to be conscious of the difference immediately. For the rest of us the effect of excellent

character fit (as opposed to merely good) is subliminal.

## Logical to Physical Font Mapping and Font Caching

Troff's model of a typesetting machine assumes a single master font for, say, Times Italic, and a lens-system which scales the character-images on that font to the chosen pointsize. Thus typeface and size are treated orthogonally.

However, for different reasons, both high-quality typesetting machines and laser or dot-matrix printers violate this model. High-quality typesetters divide the space of sizes into a number of ranges and have distinct masters for each range. Thus, if you take a 48-point Baskerville 'e', shrink it down photographically to the size of an 8-point Baskerville 'e', and compare it with the 'true-cut' 8-point 'e', you will find that the shrunk-down version looks thin and spindly in comparison to the true-cut version. On the other hand, laser and dot-matrix printers use a device-resolution so low that there has to be a different digital-master for each pointsize of a typeface. Usually, as a result, there are only a few pointsizes available, and the set of available pointsizes may vary from typeface to typeface.

To keep the following discussion intelligible, let us agree that a 'font family' is a set of related typefaces (e.g., 'Times'), that a 'face' is a specific style of character within a font family (e.g., 'Times Italic') and that a 'font' is a specific 'face' in a specific '(point)size' (e.g., 12-point Times Italic). Used this way, 'font' corresponds to what the early printers meant by that term.

What we have done, then, is to create a new mapping between the face and pointsize specified by the troff user and a physical font chosen from the set of fonts available on the target output device. This mapping, set up by the system administrator, handles the 'ranges' available on the expensive phototypesetters as well as the problem of choosing a 'best efforts' emulation of the user's specification on cheaper, cruder devices.

Within the device description file (DESC), there are multiple 'logifnt' specifications, one for each face that the user may refer to in a '.ft' request or '\f' function. A formal description of the 'logifnt' specification follows:

```

<spec> ::= logifnt <logical face>
        <mapping list>

<mapping list> ::= <mapping>
                  | <mapping> / <mappinglist>

<mapping> ::= <ps_spec>:<fontname>
              | <ps_spec>:<fontname>,
                <ps_multiplier>

<ps_spec> ::= <psize>
              | <psize>,<ps_spec>

<psize> ::= <integer>
            | <integer>--<integer>

<ps_multiplier> ::= <integer>
                   | <null>

<fontname> ::= <filename>

```

The fontname, here, is the same as the name of a file in /usr/lib/font/dev???. The psize refers to integral point sizes that might be specified by the user in a '.ps' request or '\s' function, while the ps\_multiplier is a factor to be used in conjunction with the font's width and kerning information to arrive at the actual size of each character. If the ps\_multiplier is null, then the user-specified pointsize (suitably scaled) is used as the multiplier.

Note that the ps\_multiplier allows a (very limited) way to cope with fractional point sizes -- something badly needed for dot-matrix printer support.

Note also that the user may use long fontnames instead of two-letter ones.

Consider the following example:

```
logifnt Helvlight 5-18:Helvlt/19-120:Helvltb
```

The logical font is Helvetica Light, requested with the use of '.ft Helvlight' in troff. If the current pointsize is between 5 and 18 points the physical font Helvlt is used. If the pointsize is between 19 and 120 points, the physical font Helvltb is used. These could be two 'ranges' on an expensive phototypesetter.

The mapping for a laser printer with a digital master for each pointsize in Times Roman could be specified as follows.

```
logifnt TR 8:TR08/9:TR09/10:TR10/
```

```
12:TR12/14:TR14/18:TR18/
24:TR24
```

For each pointsize in this example there is a unique font-table specified: 'TR08' for 8-point Times Roman, and so on.

With this specification, a request for 11-point type would result in a reasonable substitution. Requests for smaller than 8-point would be mapped into 8-point, while requests for larger than 24-point would be mapped into 24-point.

To maintain compatibility with old troff, there is a new troff request which allows the user to associate the style-names R, I, B, and BI with a font-family, by prefixing the family to the style. The default is 'T' (for Times, most likely), so that '.ft R' really refers to logical font 'TR'. This means that there must be logifnt specs in the DESC file for at least TR, TI, TB, and TBI.

For example the logifnt specs for the old C/A/T photomechanical typesetter would be:

```
logifnt TR 6-12,14,16,18,20,22,24,28,36:R
logifnt TI 6-12,14,16,18,20,22,24,28,36:I
logifnt TB 6-12,14,16,18,20,22,24,28,36:B
logifnt TBI 6-12,14,16,18,20,22,24,28,36:BI
```

These specs list the available point sizes and indicate that a single physical font is used for a given face at all legal point sizes.

Because devices with a unique digital master for each pointsize/face combination is likely to force troff to refer to a very large number of width-tables in one line, and also to remove any restriction on the way fonts can be 'mixed' in a line, we implemented a font caching scheme. The size of the buffer-cache can be controlled so that there is very little swapping of tables on large machines with complex typesetting requirements, yet no waste of memory when the machine has a limited address-space.

One consequence of this caching mechanism is that the concept of 'font position' has lost any real meaning. To maintain compatibility with some macro packages, we preserve the '.fp' request as a way of setting up 'aliases' between numbers and specific fontnames, with suitable defaults.

All this hacking suggested other enhancements as we went along, such as long macronames (and hence hashed instead of linear searching of requests and macronames), a list of 'special fonts'

associated with each individual regular font, character-strings (with motion and other functions) instead of the single output-codes in the font tables (supplanting much of the function of /usr/pub/eqnchar and making easy the use of 'national character sets' found on newer printers), and loading the output strings only in the post-processors and not in troff itself.

Internally, the 'cookie' (typedef tchar) that carries around a character no longer specifies the pointsize and face for the character, but instead points to an entry in a glyph descriptor table which can have as many attributes encoded there as are useful. Already it has the '.bd' and '.cs' information, and 'colour', as well as pointsize and face. Thus the time-honoured caveats in the troff manual ('The mode must be still or again in effect when the characters are physically printed.') can at last be ignored.

Adding attributes now has no real effect on processing speed, because troff simply compares the attribute-pointer in a cookie to the saved pointer for the previous cookie. Most of the time they are equal and no further processing is needed.

## A New Intermediate Language

From experience we have found that the troff (ditroff) intermediate language is too low-level for the postprocessors of some devices. Each letter is accompanied by the 'escapement' that positions it with respect to the previous letter. This is information which postprocessors can determine for themselves (and in fact they always have). Furthermore, sometimes the postprocessor needs to second-guess troff, for example when it is emulating a high-quality phototypesetter on a low-resolution device so that the user can check the line- and page-breaks before running off a final copy on the expensive device. But second-guessing is difficult if the information you receive is too low-level.

So we have rewritten the part of troff that generates output, still keeping the language very simple to scan and process, but leaving it up to the postprocessor to compute the escapements.

This change has had a dramatic effect: first, the intermediate code is far more readable by humans; second, troff runs substantially faster; and third, the postprocessors run very much faster. By the way, the intermediate files are now one-third their former inflated size.

Instead of simple motion-commands, we now pass to the postprocessors more meaningful commands, such as 'change the standard leading', 'change the indent', 'change the line-length', and, above all, 'set the following line'. This last command supplies for the postprocessor's convenience the number of 'paddable' wordspaces and the amount of 'slack' available for distribution over them. Thus, in the simplest cases, the postprocessor knows exactly what horizontal space to put out when it encounters a space, and in more complex cases the postprocessor has all the information needed to second-guess troff.

## Maintainability and Speed

As we have worked, we have taken every opportunity to make troff more modular. Thus we have a 'hyphenation module', a 'glyph information management module', and an 'output module', each of which hides as much information from the rest of the program as possible, and each of which can be tested separately. Work on simplifying the interfaces to these modules led to many opportunities to simplify the code in the rest of troff. On many days our work consisted solely of making troff more maintainable.

Curiously (or maybe not so), improving troff's maintainability seemed to go hand-in-hand with speed-improvements. At the time of writing this paper we do not know where this process is going to stop. Stay tuned for updates.

## References:

- [Liang, 1983] Liang, Franklin Mark, Word Hyphenation by Computer. Stanford University, Dept. of Computer Science. Report No. STAN-CS-83-977.
- [Bloom, 1970] Bloom, Burton H. 'Space/Time Trade-offs in Hash Coding with Allowable Errors', CACM v.13, n.7 (July, 1970).
- [Gremillion, 1982] Gremillion, Lee L. 'Designing a Bloom Filter for Differential File Access', CACM v.25, n.9 (Sept., 1982).

## Interactive Examination of a C Program with Cscope

Joseph L. Steffen

AT&T Bell Laboratories  
Naperville, Illinois 60566

### 1. INTRODUCTION

Changing an unfamiliar program means learning enough about it to at least find the code to change. Serious bugs can be introduced because the effects of changes are not well understood. This is such a daunting task for large programs that many desirable changes are not made. However, changes may be demanded by users to fix bugs or tailor the program to their needs. Thus we need tools to help us learn how a program works.

Some existing tools like `cxref` generate symbol to source line cross-reference listings. They receive surprisingly little use--probably because their output is awkward to use. Before writing `cscope`, I was a dedicated user of symbol cross-reference listings. However I often ran out of fingers to use as placeholders in the source and cross-reference listings, and there are only so many paper clips you can put in a thick listing and still remember what they are for.

Other tools like `cflow` list the function call hierarchy in a program, but once again these are little used. `Ctags` makes it easy to jump from one function to another in an editor like `vi`, so this tool combination is used by some people as a program browser. Because it is simple to use, `grep` is the most popular tool other than an editor for examining a program.

What is needed is a single tool that presents the functionality of the above tools with an interactive, friendly user interface. It should have a smooth interface to the your preferred editor so you can easily use the analysis output to make changes to your program. It should help you:

- Learn how a C program works without flipping through a thick listing.

- Locate the section of code to change to fix a bug without having to learn the entire program.

- Examine the effect of a proposed change such as adding a value to an `enum` variable.

- Verify that a change has been made in all files such as adding an argument to an function.

- Rename a global variable in all source files.

- Change a constant to a preprocessor symbol in selected lines of files.

Specifically, it should answer questions like:

- Where is this variable used?

- What is the value of this preprocessor symbol?

- Where is this function in the source files?

- What functions call this function?

- What functions are called by this function?

- Where does the message "out of space" come from?

`Cscope` is an interactive screen-oriented tool designed to meet these goals. It answers questions like these from a symbol cross-reference that it builds the first time it is used on the source files. On a later call, `cscope` rebuilds the cross-reference only if a source file has changed or the list of source files is different. When the cross-reference is rebuilt the data for the unchanged files is copied from the old cross-reference, which makes rebuilding much faster than the initial build.

### 2. EXAMPLE

The best way to describe `cscope` is to go through an example. Its most common use is on a directory containing the source of a single C program, so just change to the directory and type:

```
cscope
```

If `cscope` needs to update the cross-reference it will print messages informing you of its progress:

```
cscope: symbol cross-reference for
main.c built
input.c copied
scan.c built
/usr/include/stdio.h copied
```

Note that included header files are put in the cross-reference. After the cross-reference is ready cscope will clear the screen and display this at the top:

Cscope version 5.2

Press the ? key for help

At the bottom of the screen it will display these input fields:

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
```

If you press the ? key for help this will be displayed:

Press the TAB key repeatedly to move to the desired input field,  
type the text to search for, and then press the RETURN key.  
If the search is successful, you can use these single-character commands:

```
1-9      Edit the file containing the reference.
space bar Display more references.
>        Append the list of references to a file.
```

At any time you can use these single-character commands:

```
.      Search with the last text typed.
!      Start an interactive shell (type ^D to return to cscope).
^L     Redraw the screen.
?      Display this list of commands.
^D     Exit cscope.
```

Note: If the first character of the text you want to search for matches a command, type a \ character first.

Lets say you type getchar in the first input field and press the RETURN key:

List references to this C symbol: getchar

Searching will be displayed for a short time at the top of the screen and then this will replace it and appear above the input fields:

C symbol: getchar

File	Function	Line
1 main.c	main	9 c = getchar();
2 main.c	process	58 while ((c = getchar()) != EOF) {
3 main.c	chgstring	169 if (getchar() == EOF) {
4 input.c	getline	22 c = getchar();
5 input.c	askforchar	66 getchar();
6 scan.c	token	38 if ((c = getchar()) != EOF) {
7 scan.c	number	174 while ((c = getchar()) != EOF) {
8 scan.c	comment	292 while ((c = getchar()) != EOF) {
9 stdio.h	<global>	63 #define getchar() getc(stdin)

\* 2 more lines - press the space bar to display more \*

You can type the number (1-9) of a reference, and `cscope` will call your preferred editor to display the code surrounding that reference.

### 3. OTHER INPUT FIELDS

The input fields are grouped by function and ordered by frequency of use to minimize the number of times you need to press the `TAB` key. The preceding example showed that this input field:

List references to this C symbol:

functions as an interactive symbol cross-reference with a smooth interface to your preferred editor. This input field:

Edit this function or #define:

puts you in your editor at the beginning of the function or preprocessor macro definition. Its main use is reading code. For example, when you want to learn how a program works you often start with the `main` function. This input field lets you go directly to a function without having to first request the list of function references, visually find the function definition, and then select that reference.

As a side effect of finding a macro definition it also finds the definition of a preprocessor constant. This is useful in large programs where a constant may be used hundreds of times, so it would take a long time to find its definition by paging through the list of references.

This input field:

List functions called by this function:

gives a downward view of the function hierarchy that is helpful when looking at a high level function like `main`. This input field:

List functions calling this function:

gives an upward view of the function hierarchy that tells you what functions may be affected by a change to this function. This input field:

List lines containing this text string:

searches the source files with `egrep`, which is even faster than `fgrep` for a simple string search. The display of text strings found does not list the current function because this is not in the `egrep` output. This anomaly has not been corrected because code to both search for a string and recognize function definitions is likely to be slower than `egrep`, so it is not worth the effort to write.

This input field:

Change this text string:

solves the problem of renaming global variables or changing numbers to preprocessor constants in all source files. This is tedious to do with an editor, and error-prone with `sed` because the string being replaced may occur within other strings. After you enter the text string `cscope` will prompt for the replacement string, search for the text string, and then display the references. At the bottom of the screen will appear the message:

Select lines to change (press the ? key for help):

If you press the ? key for help this will be displayed:

When changing text, you can use these single-character commands:

1-9	Mark or unmark the line to be changed.
*	Mark or unmark all displayed lines to be changed.
space bar	Display more lines.
a	Mark all lines to be changed.
^D	Change the marked lines and exit.
ESC	Exit without changing the marked lines.
^L	Redraw the screen.
?	Display this list of commands.

Lines you have selected to change are marked on the screen with the > character:

Change "getline" to "mygetline"

```

File      Line
1>main.c   9 c = getline();
2>main.c   58 while ((c = getline()) != EOF) {
3>main.c  169 if (getline() == EOF) {
4 input.c  22 c = getline();
5 input.c  65 /* ignore next char (and lint complaint about
      getline value not used) */
6 input.c  66 getline();
7 scan.c   38 if ((c = getline()) != EOF) {
8 scan.c  174 while ((c = getline()) != EOF) {
9 scan.c  292 while ((c = getline()) != EOF) {

* 3 more lines - press the space bar to display more *

```

If you type ^D to change the marked lines this will be displayed:

Changed lines:

```

c = mygetline();
while ((c = mygetline()) != EOF) {
    if (mygetline() == EOF) {

```

Type any character to continue:

Note that the text string search finds references inside comments, whereas the symbol search does not. This lets you change global variable names inside function prologue comments and commented-out code.

#### 4. STACKING CSOPE AND EDITOR CALLS

When `cscope` puts you in the editor to display one symbol reference and you see another symbol of interest, do not leave the editor, just call `cscope` again from within the editor. In this way you can stack `cscope` and editor calls to off on a tangent, and then back up to where you were by exiting

the tangential `cscope` and editor calls.

For example, when learning a program you will probably have `cscope` find the `main` function and display it with your editor. You can visualize this as an inverse stack of calls:

```
cscope: find and display main()
  editor: display first lines of main()
```

You may then see that `main` calls an `init` function. Rather than exiting the editor to go back to `cscope`, call `cscope` again from within the editor to find the `init` function and display it with another editor call. Your stack of calls now looks like this:

```
cscope: find and display main()
  editor: display first lines of main()
    cscope: find and display init()
      editor: display first lines of init()
```

When you are through looking at this function, you can exit this second editor call, exit the second `cscope` call, and you are back in the first editor call that is displaying the `main` function. You are now back to:

```
cscope: find and display main()
  editor: display first lines of main()
```

and can continue reading the `main` function.

## 5. USER INTERFACE DESIGN

The user interface design was influenced by `vsh`, especially its visual display of `grep` output and interface between this and the `vi` editor. The primary goal was a simple, consistent, responsive user interface that would guide a novice without getting in the way of an expert. Additional goals were minimize the number of commands to remember and to eliminate as many sources of errors as possible. Much study, experimentation, and refinement was needed to fulfill these goals.

To be responsive, `cscope` provides immediate confirmation that a command had been recognized, and provides progress feedback during long operations such as building the database.

To be simple, all commands are single characters. To be consistent, pressing the `RETURN` key is needed only after the response to an input prompt or field, never after a command.

A reference line is selected by typing a single digit from 1 to 9. Using multiple digits would allow more references to be displayed at once, but would also require pressing the `RETURN` key after the last digit, which users often forget to do. The digit 0 was originally used but was dropped because users often typed the letter `o` instead.

The number of commands to remember was reduced by using form-entry input fields for the most common operations. However, this meant that commands could not be alphabetic characters. Wherever possible, commands were made the same as other tools:

Command	Function	Tools Using this Command
space	display next page	<code>vsh</code> , <code>more</code>
!	escape to shell	<code>vsh</code> , <code>more</code> , <code>pg</code> , <code>vnews</code>
^L	redraw screen	<code>vsh</code> , <code>more</code> , <code>pg</code> , <code>vnews</code> , <code>emacs</code> , <code>vi</code>
?	help	<code>vsh</code> , <code>more</code> , <code>mailx</code>
^D	quit	<code>vsh</code> , <code>mail</code> , <code>mailx</code> , <code>sh</code>

The > character was chosen for the command to append the list of references to a file because of its analogous use in the shell to redirect output to a file. When I could not find an analogy in other tools for a command I would ask some users for likely command characters and pick the most popular one.

I admire the "do what I mean" idea in Interlisp so there are undocumented alternate commands to the documented commands. For example, the RETURN and LINEFEED keys do the same thing as TAB. Since we have thousands of emacs users who are used to typing control characters, many of these are also alternate commands.

When text searching was added users naturally wanted to search for any printable character string, particularly numbers. Since numbers start with a digit, which is a command, it was necessary to add an escape character (\). Unfortunately, forgetting to type the escape character is a source of errors. I plan to change cscope to recognize terminals and workstations that have a mouse, and use it for selecting a reference instead of labeling the references with digits. This will eliminate most need for using the escape character, and allow more references to be displayed at once.

## 6. PERFORMANCE

All measurements were made on System V Release 2.

Program	Source Lines	Building Database	Symbol Searching	Database Size	Computer	Users
cscope	3,500	25 sec.	4 sec.	95K bytes	VAX 780	20
vi	20,000	9 min.	13 sec.	430K bytes	VAX 780	20
cni	370,000	12 min.	80 sec.	7M bytes	IBM 3081K	50

The time to update the database for cscope itself was 7 seconds when one file was changed, which is considerably faster than the 25 seconds needed to build the database initially. This makes it practical to use cscope on a program while it is being changed.

The search time is acceptable except for the 370,000 line program. 80 seconds is too slow for much interactive use of cscope, but still is better than looking through a foot-thick paper listing and cross-reference.

## 7. FUTURE

We are improving cscope to handle several million source lines of officially released source code. Since this code does not change frequently cscope can use a different database organization that takes longer to build but is much faster to search.

Robert R. Richards  
Chemical Abstracts Service

I. Overview

The BDAM access method for UNIX is a software package for storing files on a disk drive using raw I/O. BDAM is a term meaning Basic Direct Access Method and it refers to a method that IBM uses to store and retrieve disk files on their MVS operating systems. The BDAM method described in this talk is not a modification of the IBM method to UNIX, rather it is a new development for UNIX which provides BDAM-type features. The method can be used to control an entire disk or it can share the disk with other access methods, each controlling its own partition.

The standard UNIX I/O method for retrieving files from disk is slow when extremely large file sizes are involved. Data is stored in non-contiguous blocks (sectors) and is accessed by an inode indexing scheme which gets increasingly complex for large files. During I/O, all data blocks are buffered in internal system buffers to make a cache memory. This works reasonably well in a multi-user and small-to-medium file size environment. However, large files, in particular those which are larger than the system buffer size, can be stored and retrieved more rapidly if the UNIX file system and internal buffers are bypassed using raw I/O. A drawback of using this approach is the absence of any directory or table to indicate where the data blocks are stored. The BDAM method circumvents this problem by creating its own directory table.

The BDAM I/O access method is used to support a special data base management system developed by Chemical Abstracts Service called FIDO (Facility for Integrated Data Organization), but the method is general enough to be used in a number of applications where high speed retrieval of data is required, such as in a single user workstation environment. Besides application generality, disk hardware and UNIX software portability were also design goals. The method is applicable to optical disks as well as magnetic disks and can be used with System III UNIX, System V UNIX or other System V compatible operating systems.

The BDAM disk files are administered by a Volume Table of Contents (VTOC) which serves as the directory for the volume. The VTOC contains general information about the volume as well as file size and location information. BDAM programs have been designed to create, move and delete VTOCs. BDAM subroutines have been designed to perform the functions of opening, closing, reading, writing, adding extents, and deleting files. The open routine contains an option for creating new files. A special feature for System V applications is a no-wait option for reading and writing. The subroutines are not true system calls but are instead subroutines which use UNIX low level system calls of open, close, read, write and lseek. The routines are stored in binary form on an administrative library and linked into the executable module when application programs are compiled.

## II. Design Assumptions.

Several assumptions were made in the design of the access method:

- o The design uses UNIX and C.
- o The UNIX is either System III, System V, or a modification of either.
- o The DASD storage media is either magnetic or optical disks.
- o The method uses raw I/O.

Several design decisions were made regarding the disk layout:

- o All BDAM files begin on track boundaries. No two files are allowed to co-exist on one track.

The reasons for this are to simplify the management of disk space. A relatively small bitmap is used to administer track availability. Managing track space with a bitmap results in contiguous files with little fragmentation of the disk. File extending and management of miscellaneous tracks is easier. The alternative is to make files begin on physical sectors. This may result in less disk wastage but more file fragmentation. Retrieval times of fragmented files would be longer, and disk administration would require more overhead.

- o File lengths are allocated on the basis of extents.

Extents are an internal accounting method for adding non-contiguous lengths to files. Depending on the fragmentation of the disk this could result in more than one physical extent being allocated. From the user point of view, files consist of a collection of logical blocks and how they are allocated on the logical device is transparent to the user.

- o A file can have an unlimited number of extents.
- o The BDAM access routines perform all reads and writes beginning on physical sector boundaries.

UNIX raw I/O requires this. Designing the routines to function on sector boundaries is easier and results in efficient executable code.

- o The user programs specify file lengths, extent lengths, RBAs (Relative Block Addresses), and read and write lengths in multiples of the device sector size. These lengths are referred as "logical blocks". Logical block size is file dependent.

This is a departure from true IBM BDAM access where communication is expressed in terms of bytes. There is also potentially more disk wastage. However, our current application programs are designed to do I/O in terms of blocks and there is no desire to change this.

- o The BDAM access routines are subroutines and not true system calls.

User interface with the operating system is adequately handled with the currently available system calls. For portability reasons the current calls are used.

### III. Administrative Files

#### Volume Table Of Contents

BDAM disk files are administered by a Volume Table of Contents (VTOC). The VTOC consists of two parts, a static and a dynamic part. The static part contains the volume name and device specific information, such as the device sector size, track size, and total number of sectors. The dynamic part contains file descriptor structures, extent structures, and the track bitmap. The static part is always contained on the device. When the device is a magnetic disk, the dynamic part is contained on the device also. When the device is an optical disk, the dynamic part temporarily resides on an associated magnetic device (as a UNIX file). The reason for this is that data on optical disks cannot be erased, and the dynamic part of the VTOC requires changing as files are added to the system. When all files have been added the dynamic part is moved to the optical disk.

Without exception the first track of the device is reserved for both parts of the VTOC. Data files may not exist on the first track. The VTOC uses the first sector of this track for the static part. The dynamic part begins on the second sector.

#### Volume Index Table

A file called the Volume Index Table (VIT) is also part of the design. It contains a list of VTOCs along with logical device names, volume names, and VTOC location information. One reason for the VIT is to allow users to specify volume names instead of physical device names. Another reason is to specify where the dynamic part of the VTOC is located. The VIT is an ordinary UNIX file and there is only one per system. The BDAM access routines must open and read the VIT as well as the VTOC whenever a request is made to open a file.

When a file is opened the BDAM open routine opens and reads the VIT first. It then reads the VTOC into memory with two read calls. It reads the static part first, then it reads the dynamic part from either the same device or from a UNIX file. Admittedly, the requirement to do three opens and reads does not appear optimum, but for our application programs, data files are not opened often enough for this to be a significant performance factor.

#### File Control Block

A pointer to the File Control Block (FCB) is returned by the BDAM open routine to the application program. The FCB is a data structure which contains file related data obtained from the VTOC. It contains the file's data block addresses as well as file specific information. Space for the FCB structure is dynamically allocated during the BDAM open call; FCB space is freed by the BDAM

close routine. Application programs need only to declare pointers to the FCBs. The FCB structure type is declared in a common BDAM header file which is included in application program source files.

#### IV. BDAM Subroutines

##### Open and Close

The function of the BDAM open routine is to obtain all of the data file parameters from the VTOC, allocate space for a File Control Block (FCB), fill in the FCB, and return the FCB pointer to the application program. Refer to Figure 1 in Appendix A. In accessing the VTOC (via the UNIX low level open call on the device) for this information, the open routine maintains an internal table in order to keep a record of how many BDAM open calls have been made on the logical device by the current process. Only the first BDAM open call actually opens the device; later open calls just increment a counter in this table.

The open routine also allows new files to be created. In such cases it locks the VTOC, does a scan through the track bitmap for the necessary number of contiguous tracks, marks those tracks busy, fills in the FCB, and returns the pointer. If the number of contiguous tracks is not found, the number is divided by two and the bitmap is scanned again. An extent is automatically allocated if division is necessary. This algorithm continues until one track per extent becomes allocated. An error is returned if the file does not fit into the bitmap. Checking is also done to eliminate duplicate file names from being created.

The BDAM close routine frees up the space used by the FCB. It also decrements the counter; if the counter then equals zero, it does a UNIX low level close call on the device.

##### Read and Write

There are two sets of BDAM read and write operations. One set is used to wait for the read and write transaction to complete. This is referred to as the wait operation. There is also a no wait operation; it returns immediately. The no wait operation is functional for UNIX System V versions only because it needs to use the interprocess communication features of System V. A different set of BDAM subroutines has been designed for the wait and no wait operations.

Refer to Figure 2. The application program passes the FCB pointer along with a data buffer address and the number of blocks to be transferred to the read and write routines. The routines assemble all of the necessary extents from the FCB before doing any reads or writes. Then, the minimum number of UNIX low level reads or writes are done in order to complete the transfer. The number of blocks actually transferred is placed into the application program's return data buffer and also used as a return value.

##### No Wait reads and writes

For no wait operation a special BDAM no wait open subroutine was developed for the application program. This subroutine creates a BDAM server program. See Figure 3. The server executes UNIX low level reads and writes in response to

IPC messages received from parent BDAM read and write routines. If the length specified by the user spans more than one track and the tracks are not contiguous, the parent assembles several read (or write) requests into one message rather than sending separate messages for each request. The parent routines return immediately to the application program. The application can later determine the results of the transfer by issuing a BDAM check call. This is equivalent to doing a wait.

Refer to Figure 4. Because it is desirable for there to be only one server per process per logical device, a server is only created on the first BDAM open call for the process. A check of all allocated FCBs determines whether the logical device has been opened and, hence, whether a server exists. Each BDAM close routine does a similar check until no FCBs are found, at which time the server program is terminated by the routine. The server is a small program which contains a message receiver, a routine to check and reset semaphores, and BDAM low level read and write routines. The server stays in a continuous loop. It begins by waiting for a message from the parent. When it receives one it calls the appropriate BDAM routine to make the transfer, waits for its completion, writes the return value in terms of logical blocks transferred into the shared memory return field, resets a semaphore if one has been posted, and waits for another message.

From the application's point of view, the only difference of the no wait from the wait subroutines is the locations of the data buffer and return fields; in the wait case they can be in the application's data area but in the no wait case they must be in shared memory. To use the no wait feature the application program must first acquire a segment of shared memory and transfer its data there. The application program then passes a segment pointer to the BDAM parent routine as a parameter.

#### No-wait check

The application program which uses the no wait operation can check the status of a transfer by invoking the BDAM check routine. The check routine first reads the return field (in shared memory) and, if the value is 0, posts a semaphore and waits for the semaphore to be reset by the server. At such time it wakes up, reads the return field again, and repeats this until the return field is filled in. It may be woken up several times when several BDAM servers are processing read and write requests on different devices. When it eventually finds a value it returns that value to the user.

#### V. Conclusion and Acknowledgments

The BDAM I/O Access Method will be used in a number of applications at Chemical Abstracts Service as well for applications in the Automated Patent System being developed for the U.S. Patent and Trademark Office. This development was funded in part by a subcontract from Planning Research Corporation under Contract No. 50-SAPT-4-00319 with the U.S. Patent and Trademark Office. The author wishes to thank CAS staff members Tom Couvreur, Tom Milligan, and Rich Scott for their helpful suggestions in the design of the BDAM I/O Access Method.

## APPENDIX A

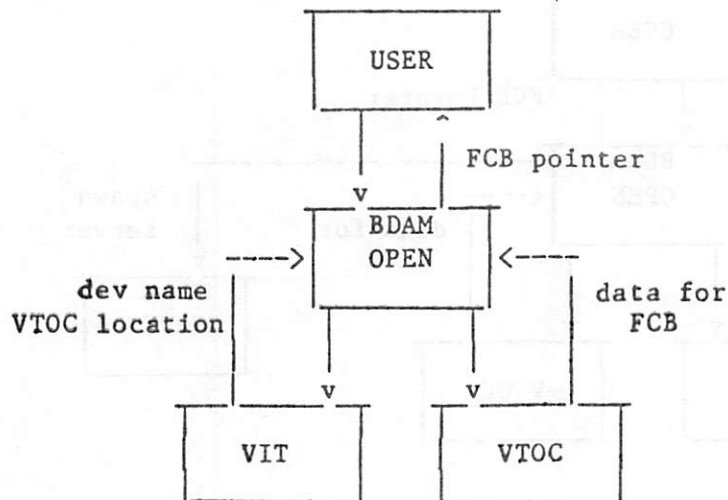


Figure 1. BDAM Open

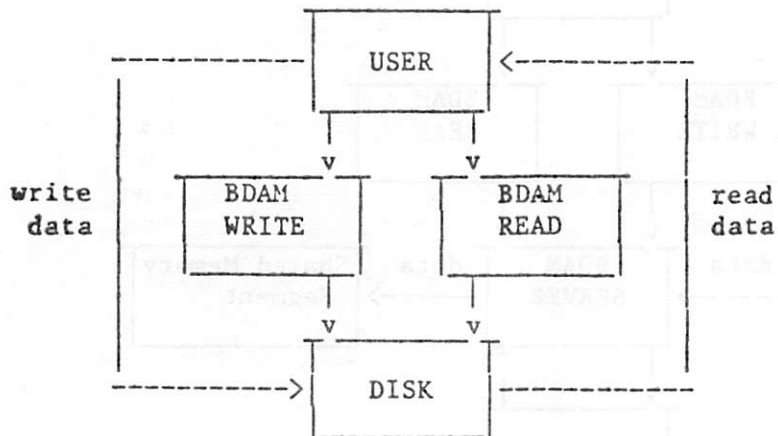


Figure 2. BDAM Reads and Writes

# APPENDIX A (Continued)

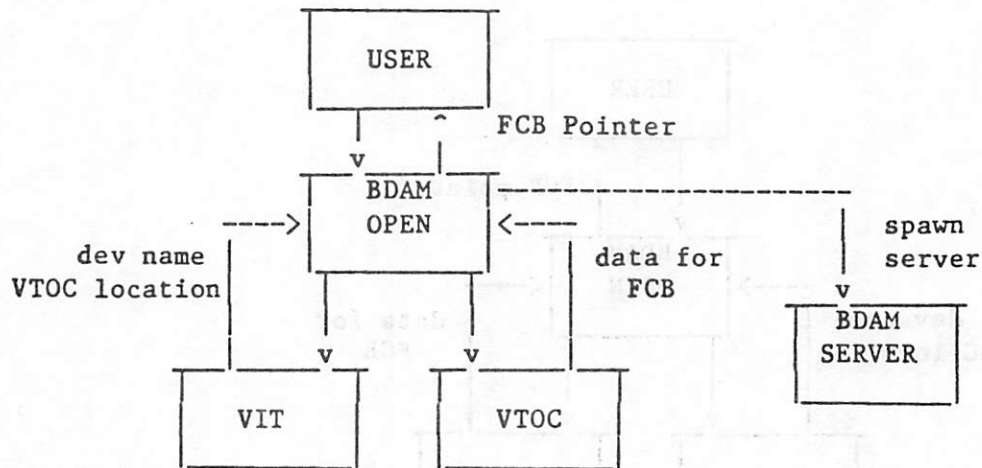


Figure 3. No Wait BDAM Open

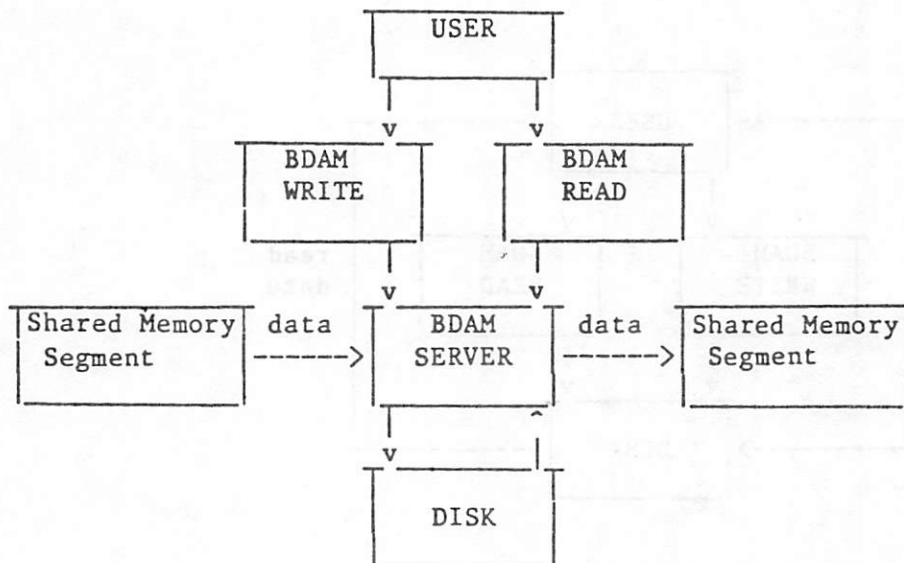


Figure 4. BDAM No Wait Read and Write

# STATUS OF THE USENIX UUCP PROJECT

Karen Summers-Horton  
Mark Horton

2843 Valcour Court  
Reynoldsburg, OH 43068  
614-864-4358  
lhnp4!cbosgd!mark

## ABSTRACT

This talk will summarize the progress of the USENIX UUCP Project. USENIX has funded the project since April, 1984. Volunteers working for the project have undertaken the enormous task of organizing the UUCP network into a manageable structure. We have now completed the first phase and have a useful map of the UUCP sites. This map is being kept up-to-date and is available to the UUCP community.

The second phase organizing the flat name space of UUCP into a more manageable, hierarchical domain has begun. A set of requirements for subdomains is in draft form and is being circulated for comment. Some of the UUCP subdomains have already been identified. Standard formats for mail interchange using domains have been determined. Public domain software to transfer mail using this new standard is under construction.

## A Parser for Electronic Mail Addresses

Peter Honeyman  
Pat E. Parseghian

Department of Electrical Engineering and Computer Science  
Princeton University  
Princeton, New Jersey 08544

### ABSTRACT

We describe **pathparse**, a tool with a practical approach to disambiguating problematic routes such as *down/bruce@asburypark*. **Pathparse** resolves syntactic and host name ambiguities with the aid of a database that describes host connections. The parser generates a set of path/user pairs; we pick the best among these according to a set of heuristics, and pass the result to a conventional mail router. With a sizable database, routes are typically parsed in a few seconds. **Pathparse** is a tool to be used within a larger mail system, and makes no special assumptions about the mail environment in which it is running.

### 1. Introduction

As the network of UNIX<sup>†</sup> hosts continues its explosive growth, and as gateways to other networks become widely used, electronic mail routing becomes increasingly complex. Two critical problems thwart conventional techniques for mail routing:

1. a mixture of right-associative (*host/user*) and left-associative (*user@host*) operators produces ambiguous routes, and
2. distinct hosts may have identical names.

The first problem interferes with straightforward parsing techniques, while the second hampers route selection and optimization.

An obvious approach to mixed precedence ambiguity is to standardize address syntax, *e.g.*, by introducing precedence rules or by rejecting routes with operators of mixed types. A straightforward solution to host name ambiguity is to impose an artificial structure on the network, *i.e.*, prohibit distinct hosts with identical names. If successful, these approaches eliminate both sources of ambiguity. Unfortunately, they require sweeping changes to a large, distributed body of software and as such, are difficult to implement or enforce.

In this paper we describe **pathparse**, a tool with a practical approach to disambiguating problematic routes such as *down/bruce@asburypark*. **Pathparse** resolves syntactic and host name ambiguities with the aid of a database that describes host connections. The output is a pair consisting of a host name and an address; the host name is one that is uniquely defined in the database, and the address is the destination relative to that host. We designed **pathparse** to be used as a tool within a larger mail system incorporating other features such as route selection and optimization.

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories

We characterize the electronic mail network as a labeled directed graph, allowing distinct nodes to have identical labels. (This corresponds to distinct hosts having the same host name.) We assume the reader is familiar with the basic terminology of graph theory, such as *vertex*, *edge*, *path*, and *path length*.

Each edge in the graph has attributes that direct route formation. One such attribute is the set of operators that may be used with that edge. Typical left-associative operators are "@" and "!", as in *user@host*; common right-associative operators are "!" and ":", as in *host!user*.

Let  $P$  be a path and let *user* be a user or service on the host corresponding to the last vertex on  $P$ . A *route* is defined recursively, as follows. If  $P$  has length 0, then the route associated with this path is simply *user*. Let  $P'$  be a path of length  $k+1$  formed by appending a path  $P$  of length  $k$  to an edge  $u \rightarrow v$ , let *host* be the label of  $v$ , let *route* be the route corresponding to  $P$ , and let  $\theta$  be an operator associated with edge  $u \rightarrow v$ . If  $\theta$  is left-associative, then a route for  $P'$  is *route* $\theta$ *host*, otherwise a route for  $P'$  is *host* $\theta$ *route*.

By convention, the host corresponding to the first vertex in a path is not shown in the route. Note that one path may be associated with several routes; this follows from the availability of several operators for a given edge. We see then that the relationship between paths and routes is not one-to-one.

**EXAMPLE:** Let us equate vertices with their labels. Consider the path

seismo  $\rightarrow$  ucbvax, ucbvax  $\rightarrow$  cmcl2

If each edge has the left-associative operator "@" and the right-associative operator "!", then this path has the following routes:

ucbvax!cmcl2!user  
ucbvax!user@cmcl2  
cmcl2!user@ucbvax  
user@cmcl2@ucbvax

Note that the second and third routes combine operators of both types, and also correspond to the path

seismo  $\rightarrow$  cmcl2, cmcl2  $\rightarrow$  ucbvax

while the first and fourth routes lack this ambiguity. Thus, since distinct paths may yield the same route, the relationship between paths and routes is many-to-many.

### 3. Parsing routes

The goal of parsing is to find all paths in the graph that correspond to a given route. Given a route devoid of operators, the parse is trivial: we derive a single vertex, our source host. For a route containing only operators of the same type, the parsing task consists of mapping the sequence of host names to edges in the graph. If the operators are all left-associative, this mapping is performed from right-to-left on the route, otherwise from left-to-right.

When a route is composed of host names separated by operators of mixed types, we have a choice for the first edge of the path: we may either follow an edge from the source host to the left-most host, or an edge from the source host to the right-most host. We try both choices: given a syntactic ambiguity in parsing the route, we rely on host connection data to tell which choice is semantically viable. That is the fundamental rule of the parser.

If the database query corresponding to one of the choices succeeds, we delete that host and operator from the route and parse the remainder recursively, changing our notion of the source host to the host we deleted. Should both choices succeed, we recursively parse the remainder of both edge choices, hoping to resolve the ambiguity at a deeper level of the parse.

### 3.1. Candidate parses

The result of parsing a route is a set of candidate path/user pairs. Initially, the candidate "path" is the source host and the candidate "user" is the rest of the route. If we are able to parse a route completely, i.e., find a set of edges that leaves us with a user segment containing no operators, we save this path/user pair for later consideration. Given a set of edges that constitute a partial path in the graph, and a remaining route to be parsed, we may find that the choice presented by selecting the left- or right-most host fails the database query. We also save this path/user pair, although in this case the path is incomplete and the remaining route contains operators. At a later stage, we use additional criteria to evaluate the candidate routes and make a choice. Before describing these criteria, we cover the parsing algorithm in detail and work through an example.

### 3.2. Parsing details

From the rules for formation of a route, it is not possible to build a route in which a right-associative operator follows a left-associative one. Thus all routes take the form of a set of hosts connected by right-associative operators, then a user, followed by a set of hosts connected by left-associative operators. If neither set of hosts is empty, the route is ambiguous. As described above, such a route may specify a path that begins with an edge to the first named host, or one that begins with an edge to the last named host. To resolve this ambiguity, we consult a database that describes the graph. The database supports a predicate `isalink(host0, host1)`, which returns `true` if the database contains an edge  $v_0 \rightarrow v_1$  with labels `host0` and `host1`, respectively, `false` otherwise.

The basic parsing algorithm is a recursive backtracking one. Suppose that a given route **R** is composed of both left- and right-associative operators; for simplicity let us assume that **R** is the route `down!bruce@asburypark`, and suppose the first host on the path is labeled `princeton`. We first consider the path composed of the edge `princeton→down`, i.e., we try the right-associative operator. Should the query `isalink(princeton, down)` return `true`, we subject the route **R'** = `bruce@asburypark` to further parsing, which entails querying for `down→asburypark`. If this query succeeds as well, then we have

`path = princeton→down→asburypark; user = bruce`

After a successful parse, the path and user are saved for later processing.

After decomposing a route according to a right-associative precedence, we try the left-associative parse. In the above example, this entails querying `princeton→asburypark` and parsing `down!bruce`. Again, we save a successful parse for later processing.

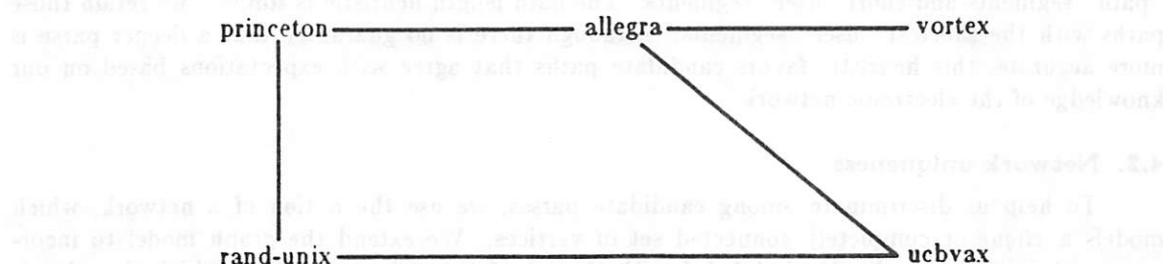
`path = princeton→asburypark→down; user = bruce`

When an edge query returns `false`, the successfully parsed portion of the path and the remainder of the route are saved for later consideration.

#### EXAMPLE:

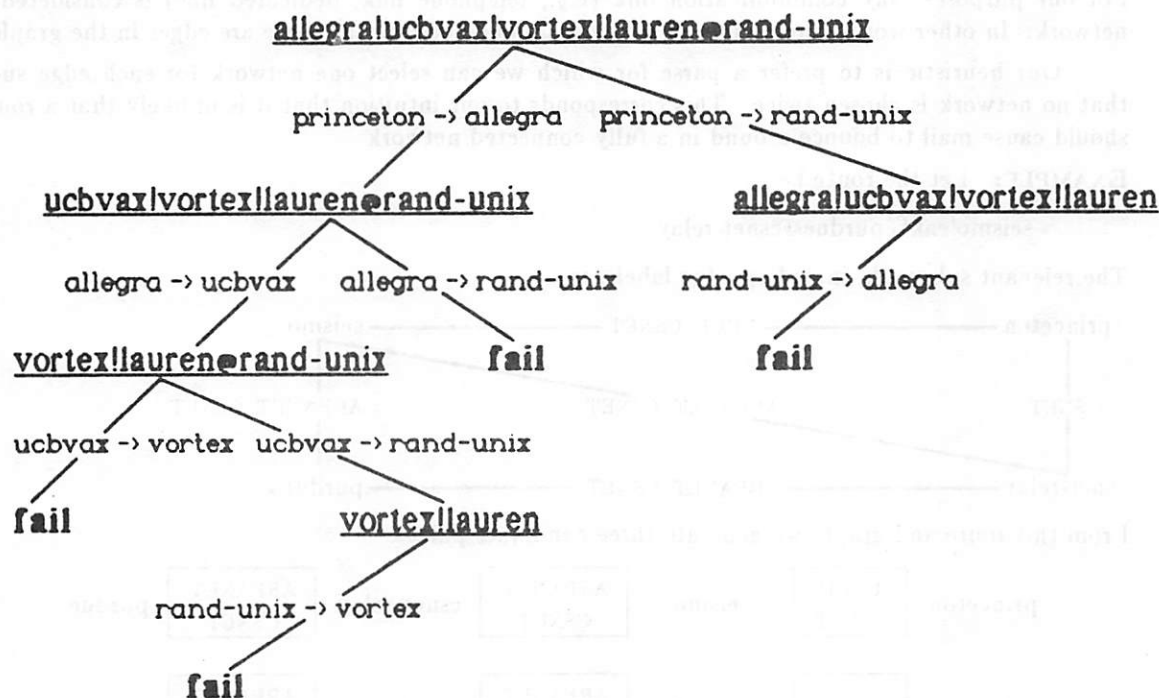
`allegro!ucbvax!vortex!lauren@rand-unix`

The relevant portion of the graph for this example is



For simplicity, we assume the graph is symmetric.

The following figure represents the recursion tree followed by the parser.



Edges of the tree are labeled by database queries. Leaves of the tree are points at which path/user pairs are saved. Thus the following four pairs are saved:

$path = princeton \rightarrow allegra \rightarrow ucbvax; user = vortex!lauren@rand-unix$   
 $path = princeton \rightarrow allegra \rightarrow ucbvax \rightarrow rand-unix; user = vortex!lauren$   
 $path = princeton \rightarrow allegra; user = ucbvax!vortex!lauren@rand-unix$   
 $path = princeton \rightarrow rand-unix; user = allegra!ucbvax!vortex!lauren$

#### 4. Choosing among candidate paths

To choose a path, we apply three heuristics: path length, network uniqueness, and common prefix.

#### 4.1. Path length

Path length is a measure of the depth of the parse: deeper parses are characterized by long "path" segments and short "user" segments. The path length heuristic is simple: we retain those paths with the shortest "user" segments. Although there is no guarantee that a deeper parse is more accurate, this heuristic favors candidate paths that agree with expectations based on our knowledge of the electronic network.

#### 4.2. Network uniqueness

To help us discriminate among candidate parses, we use the notion of a network, which models a *clique* or completely connected set of vertices. We extend the graph model to incorporate edge labels: each edge is labeled with the set of network names from which the edge is derived. An edge from no network, *e.g.*, a UUCP connection, is given a unique label.

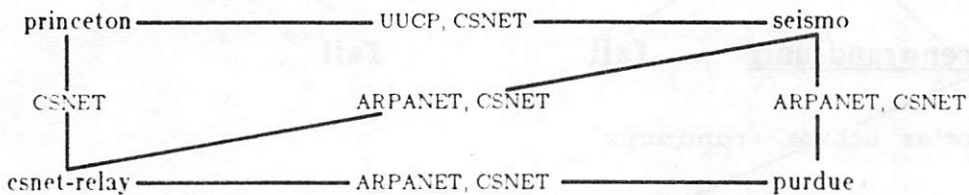
By definition, the hosts defining each edge must communicate on at least one network. These include private local area networks as well as formal networks such as ARPANET and CSNET. For our purposes, any communication link (*e.g.*, telephone link, dedicated line) is considered a network. In other words, there may be as many unique networks as there are edges in the graph.

Our heuristic is to prefer a parse for which we can select one network for each edge such that no network is chosen twice. This corresponds to our intuition that it is unlikely that a route should cause mail to bounce around in a fully connected network.

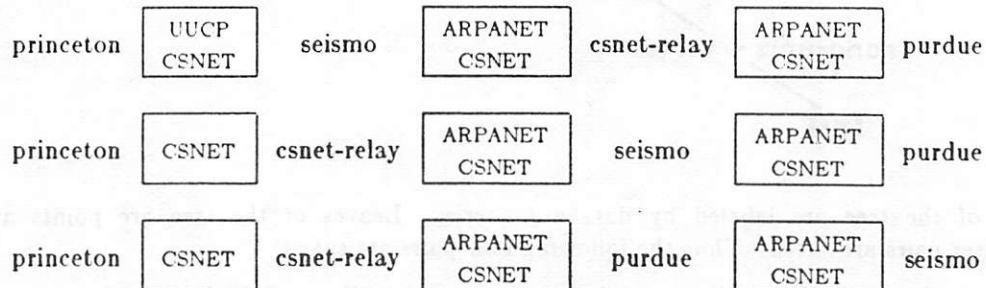
**EXAMPLE:** Let the route be

seismo!cak%purdue@csnet-relay

The relevant subgraph, including edge labels, is



From this route and graph, we generate three candidate parses:



Note that only the first parse admits the selection of a set of distinct network labels, and thus is preferred.

This network selection problem can be formulated as an instance of bipartite matching, and thus can be solved in quadratic time.\* In our case, we let  $U$  be the set of edges in the path and  $V$  be the set of network names. We connect an edge and a network whenever that edge is labeled by the network. A maximal matching in this graph corresponds to the optimal network label selection.

\* See, *e.g.*, Papadimitriou and Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, 1982.

### 4.3. Common prefix

Even after applying the above heuristics, there are routes for which multiple parses remain. For such routes, we are faced with the impossibility of semantically disambiguating the possible paths. Given two such paths, we merge them by constructing a new path that follows the common prefixes of the candidate paths up to the vertex from which they diverge. For the user information, we use the route that remains after stripping off the hosts on the common path. The hope is that the host at the end of this path is better equipped to handle the remainder of the route. If the paths diverge immediately, leaving the empty path, we are forced to reject the route.

#### EXAMPLE:

```
seismo!ihnp4!unknown!user@xerox
```

We must choose between two candidate parses:

```
path = princeton→seismo→ihnp4; user = unknown!user@xerox
```

```
path = princeton→seismo→xerox; user = ihnp4!unknown!user
```

Neither parse presents an advantage, so we truncate to their common prefix, `princeton→seismo`. We know that the route passed through `seismo`, so we expect that the remainder, `ihnp4!unknown!user@xerox`, can be more successfully parsed there.

### 5. Host name ambiguity

Were it impossible for distinct hosts to have identical names, mail routing would entail nothing more than retrieving the shortest path to the last host and routing accordingly. However, in loosely structured networks such as USENET, there can be no guarantee of host name uniqueness. Thus, we reverse the parsing process and route to the last vertex on the path that has a unique host name. Of course, there is no guarantee that the vertex we choose for this purpose is in fact uniquely named, but as long as our shortest-path database has a route to this vertex, we are guaranteed a unique route to the host.

### 6. Optimization

The result of parsing is a path in the graph and a user at the final vertex of the path. We now have two choices for reconstructing a route: we may either route to the first host on the path, which is a neighbor of the source host, or we may route to the last host on the path. In the former case, we do not change the original route, we simply use the parser to resolve the ambiguity of mixed operator types. In the latter case, we attempt to optimize using a shortest-path route that may be a substantial improvement over the original route. The decision to optimize or not is left to the applications user.

### 7. Database structure

The edge database and the shortest path database are built by Bellovin and Honeyman's **pathalias** program, a public-domain program that has been distributed on USENET. Input data for **pathalias** are also available on USENET, as well as from cooperative site and network administrators. While the database can never completely describe the graph of the interconnecting networks, in our experience it suffices to have complete connection data for the local host; this minimal amount of data can resolve most ambiguities. Of course, the more complete the input data, the better the routes generated by **pathalias** when it computes shortest paths.

We modified **pathalias** to allow declarations of "private" host names, i.e., those for which host name conflicts can be accommodated, and to create a file listing host-to-host connections, which we use as our edge database. At present, we use binary search to scan the database.

## 8. Complexity of parsing

The worst-case time complexity of the basic parsing algorithm is exponential in the number of operators. To be precise, let the number of operators be  $n$  and let the number of left-associative operators be  $l$ . Then the number of possible parses is  $\binom{n}{l}$ . This formula achieves its maximum when  $n = 2l$ ; from the Stirling approximation, this gives us  $\frac{4^n}{\sqrt{\pi n}}$  parses in the worst case. Nonetheless, we have yet to find a route that generates more than a handful of candidate parses. This is due to two factors.

First, left-associative operators tend to be used in implicitly routed networks such as ARPANET. Consequently,  $l$  tends to be one or two at most, so that  $\binom{n}{l}$  is no worse than quadratic in practice.

Second, every time an edge query fails, a large subset of the set of possible parses is rendered unreachable by our algorithm. Since the underlying graph is sparse, many edge queries fail. Thus we do not examine most of the possible parses for a given route.

## 9. A deception

To simplify the presentation, a slight deception has been carried throughout this text. Routes are most often derived from messages that have already traversed the graph; our goal is to generate a proper return path. Thus it is appropriate for us to identify a path directed *toward* the local host, rather than away from it. Accordingly, while we have claimed to query the database on edges such as *princeton*→*seismo*, the query is actually made on the edge *seismo*→*princeton*.

## 10. Performance

With a large database, describing 4,441 hosts and 16,984 edges, routes are parsed in less than two seconds on a VAX† 11/750. However, our methods for accessing the edge database leave something to be desired, requiring 100 or so disk accesses to parse a single route. Thus a parse takes about five seconds of real time. We are experimenting with other access methods, in the hope that both the real time and CPU time statistics can be significantly improved.

## 11. Conclusion

We have described a technique for parsing ambiguous electronic mail routes. Our algorithm relies on semantic information describing the network, rather than rigid precedence rules. The success of the parser is due to its ability to make decisions based on the real structure of the network, not an idealized notion of that structure.

The parser makes no assumptions about the particulars of the mail system in which it is embedded. We feel that this direction, building high-performance tools that co-exist with software in place, is preferable to the ideal of replacing all UNIX mailers with ones that conform to complex standards.

## 12. Acknowledgements

We thank Stephen C. North for many fruitful discussions, Fook-Luen Heng for suggesting the counting argument in Section 8, and Andrea LaPaugh for suggesting the matching algorithm in Section 4.2.

---

†VAX is a Trademark of Digital Equipment Corporation

# Automatic Forwarding of Mail in CSNET

Michael T. O'Brien

Bolt, Beranek and Newman

10 Moulton St.

Cambridge, MA 02138

(617) 497-2426

<obrien@csnet-sh.arpa>

decvax!bbnccalobrien

## ABSTRACT

CSNET has recently implemented Phase III of the User Name Server, which is a database of information about users of CSNET, coupled with connection-based and mail-based interrogation and update facilities. Implemented using inverted indices, the User Name Server database includes an entry for the network mail address of each registered user. The Phase III software allows definition of "nicknames", which are resolved to addresses. Mail sent to nicknames is automatically forwarded if the recipient changes his mailbox and informs the User Name Server.

## The User Name Server Local Agent

CSNET [Comer, Douglas. "The Computer Science Research Network CSNET: A History and Status Report", *Communications of the ACM* (New York) 26, No. 10 (October 1983): 747-753.] has recently implemented Phase III of the User Name Server [Landweber, L.H., M. Litzkow, D. Neuhengen and M. Solomon. "Architecture of the CSNET Name Server," (New York) SIGCOMM '83 Symposium (March 1983): 146-153.], developed at the University of Wisconsin. The User Name Server is a database of information about users of CSNET, coupled with connection-based and mail-based interrogation and update facilities. Implemented using inverted indices, [Knuth, D., *The Art of Computer Programming: Vol. 3, Sorting and Searching*. Addison-Wesley, Reading, Mass. (1977). See section 6.5.] the User Name Server database includes an entry for the network mail address of each registered user.

Users running 4.2BSD interact with the User Name Server central database through a local agent program called *ns*. This program accepts queries from the user and, after sending those queries to the Query Interpreter on the CSNET Service Host, presents the results to the user, as shown in Figure 1. Note that in this example, the query does not match my name, but

```
% ns
Welcome to the CSNET Name Server
(Type "help" if you need any)
> whois obrian
There is 1 match to your query:
-----
Name:
    Michael T. O'Brien
Account:
    obrien,csnet-sh,bbn
Ident:
    3111
Mailbox:
    obrien@csnet-sh
Phone:
    (617) 497-2426
Address:
    Bolt, Beranek & Newman
    10 Moulton St.
    Cambridge, MA 02238
Misc:
    obrian
    obrien
    csnet-staff
> bye
Goodbye
%
```

Figure 1

that I have included a common misspelling in the

*Miscellaneous* category of my User Name Server database entry so that queries of this sort will succeed. This interaction demonstrates the use of the user agent on systems that are connected to the Internet (including those connected via CSNET X25Net), and can therefore query the central database immediately. CSNET PhoneNet sites use the same user agent program, with the same syntax, but their queries are encapsulated in mail messages that are sent to the CSNET Service Host. Query results are mailed directly back to the user issuing the query.

### Nicknames

In Phase III of the User Name Server, the *ns* local agent program has been augmented to allow users to define "nicknames", which are really queries on the CSNET User Name Server central database. The queries are resolved to network addresses, which are then used to deliver mail addressed to the nicknames. If mail to such a nickname is rejected, the central database is re-queried automatically by the user's local mail system to see if the address in question has changed. If so, the mail is resent to the new address. The results of queries generated by nickname definitions are cached at the local host where the nickname is defined, thereby limiting the traffic in queries to a) the case of initial nickname definition, and b) re-queries generated by failed mail to a nickname.

The implication is that as long as users keep their registration information in the User Name Server central database current, mail can follow them around the network, with no necessity for per-system aliasing to do forwarding. In addition, the user controls the forwarding personally, as the only manual operation involved is his interaction with the User Name Server in updating his registration.

### Architecture

Previous versions of the User Name Server software have included only the local agent program, *ns*. Phase III of the User Name Server software includes a suite of programs, of which only *ns* is called directly by the user. The other programs are invoked automatically, and support the nickname query handling and mail forwarding functions of Phase III. Figure 2 shows the complete architecture of the Phase III User Name Server system.

### Mcatch

A user defines a nickname in *ns* in terms of a query, e.g., `nick foo = whois obrian` which defines *foo* as a nickname whose value is the result of the query `whois obrian`. For CSNET PhoneNet sites, where queries and their resolution are transferred by means of mail messages, some mechanism must be used to extract query results from incoming mail, and process them. This function is served by the program *mcatch*, which receives mail sent to either *forwardersite* or *resolveresite*. These two aliases, which must be established in whatever fashion the local mail transport system supports, cause *mcatch* to be invoked upon receipt of mail to these aliases. *Mcatch* examines the mail and, by parsing the header, determines which of the two programs, *resolver* or *forwarder*, should receive the mail.

### Resolver

The *resolver* program receives mail consisting of query results generated by nickname definitions in *ns* at PhoneNet sites. It installs the results of the query in the database containing the nickname definitions, which to this point have been marked as *pending*. As noted below, *resolver* also resubmits any messages whose address lists have been completely resolved by the query result.

### Filter

*Filter* is a program whose input syntax imitates the behavior of the real mail transport interface on the local system. All mail originating locally is sent to *filter*, which parses the header and looks for valid nicknames in the destination fields. If it finds any, it rewrites the header, expanding the nicknames, and inserting a new `Sender: forwardersite` field. It then sends the mail using the real mail transport system. Mail addresses created by *filter* as the result of nickname expansions carry the CSNET ID number of the recipient, which is a unique identifier in the CSNET User Name Server central database. Originally, it was intended that a separate field be used for this, but it was found that there are mail systems that routinely discard fields of this sort. This ID must be maintained with the message, as automatic forwarding depends upon it.

If there are unresolved nicknames (that is, nicknames whose associated queries have not yet been answered), the mail cannot be processed. In that case, the mail is queued, waiting for the response to the query. When the query is received, *resolver*, which receives such query

replies, scans for mail depending on that query reply, and resubmits it if there are no more outstanding queries upon which it depends.

### Forwarder

If mail is returned from some other system owing to an error, it will be returned to *forwarder* if a nickname was used in its original composition, because of the `Sender:` field inserted by *filter*. Such mail will be handed to the *forwarder* program for resolution. *Forwarder* parses the body of returned mail messages, extracting the failed address. It checks to see if there is a CSNET ID in the comment field of the address, and if so, it issues a new query to the User Name Server central database based on that ID number. If the address returned as a result of this new query is different from that in the message, *forwarder* carries out the following steps:

- 1) It extracts the original message from the text of the failed-mail message.
- 2) It rewrites the message using the new address and resubmits it to the mail system.
- 3) It resubmits any other failed messages that may be waiting resolution of that query.
- 4) It finds all nicknames on the local system that depend on this CSNET ID and updates them to reflect the new address.

### Databases

In Phase III, the User Name Server software maintains two databases on the local host, the system database and the user database. These are used to hold the two pieces of information that constitute a nickname definition. The user database contains the nicknames defined by each user. These nicknames are associated, not with a mailbox, but with the CSNET ID associated with the mailbox. The system database contains the mapping from CSNET ID to mailbox. This split means that when a user on another system changes his mailbox, only the system database on the local machine needs to be updated to correct all the nickname definitions on the system.

The databases are also used for other purposes. The user database contains the current User Name Server entry for every user on the system. This allows a user to correct his entry without having to fetch the old entry from the User Name Server central database. Also, at CSNET PhoneNet sites, the user database contains a map of the header of each message which is

queued pending resolution of a nickname query. In such cases, the system database contains the association between the query and the message map. The general rule is that messages and nickname definitions — things that are specific to users — are stored in the user database, while items related to queries are stored in the system database.

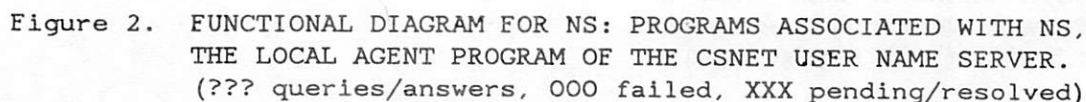
It should be noted that while nickname definitions are specific to individual users in the sense that only the *owner* can change them, any other user on the system can use such a nickname in a message address by specifying the name of the owner as well as the name of the nickname.

### Implementation

Currently, the User Name Server software is designed to be used with either *sendmail* or *MMDF* running under 4.2BSD. An upcoming release of the *MMDF* system, known as *MMDF 2*, is also supported. The dependencies lie in those parts of the system that must understand the format of messages coming from the mail system, and which must be able to parse and re-issue the syntax and grammar of the submission interface to the mail system. *Filter* must replace the submission section of the mail system, since it must scan all outgoing mail for possible use of nicknames. *Forwarder* must be able to pick apart failed-mail messages generated on other systems, and both *forwarder* and *resolver* must be able to interpret query replies in messages from the User Name Server central database. The latter function, of course, is of importance only on CSNET PhoneNet sites.

The limited number of mail systems whose failed-mail messages can be understood by *forwarder* is perhaps the most serious limitation of the system. A template facility for describing the format of failed-mail messages generated by other systems would be a large improvement. Of course, such failed-mail messages must contain the full text of the failed message, or forwarding is impossible.

The program is expensive in CPU time, particularly as it does nothing at all if no nicknames are used in the addressing of a message. Cutting the processing time here would be a large improvement.



# Notesfiles: Why You Should Use Them

Raymond B. Essick IV

Department of Computer Science

University of Illinois at Champaign-Urbana

Urbana, Illinois 61801

(217) 333-7937

## ABSTRACT

The UNIX Notesfile system is a discussion-oriented, computer bulletin-board system with a number of advantages over other computer bulletin-boards and mailing-lists. Notesfiles are useful in many applications besides USENET. Notesfile applications include problem report filing, mail processing, project workbooks, and automatic logs. This paper presents an overview of the Notesfile system, its compatibility with other UNIX tools, usability of the Notesfile system, and Notesfile's features.

## Overview

The Notesfile system provides a single tool capable of handling group discussions, mailing lists, individual mail, automatic problem reporting, and logging. Each of these applications uses a common storage format and common presentation mechanisms. A common misconception on USENET is that the Notesfile system was designed with the goal of competing with the "A-news" and "B-news" systems. This paper highlights some of the other strengths and applications of the Notesfile system.

The Notesfile system provides a flexible method for coordinating computer-based discussion forums on many topics. Discussions are grouped according to broad topics. Individual remarks are grouped with other remarks in the same discussion.

A Notesfile system contains a number of "notesfiles". Each notesfile contains many "discussions". The discussions in a notesfile center around a basic topic; typical topics include "problems", "news", "bicycles", "unix-wizards", and "policy". Discussions within a notesfile concern particular aspects of the notesfile's topic. In a "unix-wizards" notesfile, discussions might include politics of UNIX licensing, debates on the relative merits of 4.2 BSD and System V, and device driver queries. Each discussion starts with a "base note". "Responses" are the additional articles in a discussion. "Base notes" typically pose a question or state an opinion; "responses"

typically answer the question, ask a related question, or offer opposing opinions.

This multi-level structure allows users to maneuver through a notesfile in a coherent manner. Particular discussions can be isolated for special attention, relevant articles can be located, and unrelated information can be excluded. Each of these activities requires minimal keystrokes.

The remainder of this paper discusses various aspects of the notesfile system: interfaces to the standard UNIX toolkit, usability, Notesfile features, typical Notesfile applications, and a summary of Notesfile's advantages over other bulletin board systems.

## Tool Compatibility

Notes provides a set of programs to interface its database to other tools in the UNIX toolbox. Interfaces exist to accept input from pipes, act as mail filters, and provide user-callable subroutines for input to notesfiles. Other Notesfile programs manage outputting portions of the Notesfile system in various formats.

## Pipe Fitting

UNIX pipes provide a general mechanism to feed the output of one program to the input of a second program; the output of the second program can in turn be connected to the input of a third program.

Program output can be sent as input to the "nfpip" program which accepts text and places it in a notesfile. Titles, "director flags", and other options are settable as options to the nfpip program. Nfpip collects its input and generates a "base note" in the specified notesfile.

## Mail and Mailing-Lists

Computer mail is one way to manage discussions between a small number of people. Notes provides the "nfmail" program to place incoming mail in a notesfile. Nfmail examines the "Subject" line to determine whether the letter represents the beginning or continuation of a discussion; appropriate placement of the letter (as a base note or response) depends on whether the "Subject" line begins with "Re:" and a matching discussion can be found.

"Nfmail" accepts RFC-822 formatted mail and places letters into a notesfile. Information gleaned from the "Subject" line is used to establish a note/response relationship with articles already in the notesfile. The "From:" header line is parsed to determine the author of the article and the resulting information is used when entering the letter in the notesfile. Headers are not stripped from the letter unless requested; some headers (such as Subject and From) are never removed.

Notesfiles trivially become a mail sending/reading mechanism. Use an aliasing mechanism to have incoming mail directed to the "nfmail" program. While reading mail (via the sequencing mode of notesfiles), replies are initiated with a single keystroke. ("p" for personal-note; the name is a carryover from the PLATO Notesfile system.) A single line in the ".mailrc" file will generate carbons of outgoing mail and direct it to your notesfile. This allows users to maintain both sides of a mail conversation.

The Notesfile note/response structuring of discussions provides quick access to previous letters in a discussion. Referring to older letters requires only a few keystrokes.

Nfmail is useful for receiving ARPAnet mailing lists. A site only has to receive one copy of any mailing list, all local users then read the notesfile. This saves in duplicated network traffic, disk space, and machine load. The individual letters of the mailing list are grouped by discussion. Concurrent discussions are separated and it is much easier to follow the different topics.

## User Subroutines

Notes provides the programmer with a set of subroutines useful for logging information in a notesfile. One subroutine, "nfcomment", merely leaves a message in a notesfile. A second subroutine, "nfabort", generates a core image of the executing process, places it in a specified file, and records a message in a notesfile announcing this fact.

Nfcomment accepts text to place in a notesfile in one of two forms: a string supplied in the subroutine call or text collected from the standard input of the process. This text, combined with title, director and anonymous status, is placed in the notesfile specified as a parameter in the nfcomment subroutine call. A typical use of this routine is providing developers with an easy, built-in method of reporting problems with and suggested changes to their systems. The developer scans this logging notesfile on some regular basis for new comments. When no new comments are present, the "sequencer" skips over the notesfile. Monitoring a low-traffic notesfile in this manner involves very little effort.

Nfabort functions along lines similar to the nfcomment subroutine. The text to be logged in the notesfile must be supplied as a parameter to the nfabort subroutine call. Calling nfabort results in a core image of the current process being placed in a specified file. The current process ID is appended to the name, allowing several processes to leave core images without overwriting other core images. The name of the file containing the core image is appended to the text placed in the notesfile.

This function allows programmers to perform internal consistency checks, and have the aftermath of catastrophic errors placed somewhere convenient. Convincing users to save core images and other debris generated by program failures becomes unnecessary. The post-mortem information is available in a known place and a message detailing the conditions at the error point are in a notesfile. Regular sequencing through such notesfiles is painless and intrudes only when an error occurs and is logged.

## Usability

Notes is a highly interactive system. Most commands are single keystrokes; only a few commands require extra information. Moving around the notesfile data base is fast and simple. Notesfile's screen-oriented user interface typically

presents an entire screen of new information with one keypress.

Notesfile's note/response structure makes discussions easier to follow. Systems such as "B-news" and mailing lists intermingle discussions. Following several discussions at once is difficult; Notes' presents one discussion at a time. Commands to skip the rest of a discussion, save an entire discussion, or forward it to another notesfile are all single keystrokes. Following discussions is easier with notesfile's structuring than with systems like B-news or mailing-lists.

## Features

### Access Lists

Notes provides a flexible access restriction mechanism. Four types of permission exist: director, read, write, and answer. Arbitrary combinations of these permissions can be granted to users, groups, and systems. Some permissions imply others; write permission implies answer permission, director permission implies complete access.

Permissions are granted by first seeing if a user has explicit permissions in the notesfile. If so, these are used as his access rights. If not, the group permissions are examined. The user is granted the access rights of his current group. (Under 4.2 BSD, a user belongs to a set of "access groups"; a user is granted the union of the access rights for these groups.) If one or more of his groups match, the "Other" entry is not used. There is a special group, "Other", which matches all groups not explicitly listed in the access list. The user gets the access rights of the "Other" entry if his group does not appear in the list.

System entries are for networking and allow networked notesfiles to be made reasonably secure by allowing access only to specific sites.

### Transmission Costs

Notes provides a mechanism to maintain multiple instances of a notesfile in synchrony. Each instance of the notesfile is located on a different host. Additions to each copy of the notesfile are transmitted to neighboring machines. These additions are in turn re-transmitted until all copies of the notesfile have copies of the new data.

Notes maintains fixed size headers for each note and response. This information includes author, originating system, time of

writing, and time received locally. A few status bits and several strings provide routing information to prevent transmitting articles over links to a remote system which probably has the article.

The Notesfile system's reduced header information reduces the information transferred between systems. Overhead per article is approximately the size of the note header; this is currently 240 bytes. Frequently the overhead is smaller since many fields are not filled to capacity.

Any compression scheme, such as the UNIX compact/uncompact programs, would result in still further reductions in the volume of network traffic. Compression of messages is trivially implemented by modifying the command sequence used to transmit updates as it appears in the "net.how" file. (Putting compression into the notes code violates the KISS principle.) An entry for a link using compression might be:

```
somesite:x::compact | rsh somesite uncompact  
      \ | nfrcv %s %s
```

### Transmission Behavior

Notes uses a passive networking mechanism. New notes and responses are not immediately sent to remote sites. Periodic execution of the notes transmission program, "nfxmit", scans notesfiles for new articles, places them in a single batch and sends them to a remote system. The actual transmission method is selectable by modifying a table and ranges from the queued "uux" of UUCP to the immediate "rsh" mechanism using sockets. Nfxmit watches return codes to determine if the transfer was successful. If unsuccessful, the appropriate timestamps are not updated and the same articles (plus any further new articles) will be transmitted during the next nfxmit run.

One analogy of this system is that of a notes reader who requires articles to be presented in a special format. This reader frequently scans his list of articles and only marks as read the ones he comprehends. The reader is the remote system and his comprehension criteria is successfully installing the new articles in his copy of the notesfile.

This passive approach to networking results in a window for users to recall articles with typographical errors, inappropriate articles, or to change the wording of their comments.

Each notesfile comprises four UNIX files. Two of the files contain index information, the third file contains note and response text. The fourth file contains the access list. By consolidating the text and headers into a small number of files, fragmentation problems are reduced. Instead of each article taking an integral number of disk blocks, as with one-file-per-article systems, articles are consolidated into the single text file. Disk blocks lost to fragmentation in this implementation are limited to one; the last block of the file is the only "partial" block.

The four file implementation improves access times to different articles. Because all needed files are already open, accessing an article consists of nothing more than a seek and several reads. The expensive open system call is avoided. With inexpensive access to the notesfile, searches for articles by specific authors or with specific titles are fast.

The fixed size of note and response index records provides fast calculation of the location of the desired information. Retrieving a specific article consists of a multiplication, a disk seek and a disk read.

### Typical Notesfile Applications

Notesfiles are used in many applications at UIUC and other installations. Common uses include problem reporting, project workbooks, research group discussion forums, mail handling, and automatic logging.

### Problem Reporting

Computer systems typically provide a mechanism for reporting problems encountered by users. These mechanisms range from special commands and subsystems to mail aliases redirecting comments to appropriate staff. A "problems" notesfile is useful for both types of problem reporting and consolidates the reports in a single location.

Simply typing "notes problems" gets the user to a notesfile for recording problems. The user can then enter a note describing his problem, search for other notes describing similar problems encountered by other users, and find out if a solution exists and is available.

An alternate way to record a problem is via a mail alias for the mailbox "problems". Mail to the "problems" mailbox can be delivered as input to the "nfmmail" program (with

appropriate arguments). Nfmmail places these complaints into the same database of problems and solutions used by the "notes problems" command.

If the user community is allowed to read the "problems" notesfile and enter notes or responses, the result is that users often point out workable solutions before a staff member has had time to read the original complaint. This situation provides faster answers to problems, makes the user with the problem happy, and reduces the load on system staff; the solutions to the problems also become common knowledge. Problems and their solutions can be kept for future reference.

With problem reports consolidated in a common location, the system staff can reduce duplicate effort; messages of the form "are you fixing this" and "no, but John might be" don't occur as frequently. Backlogged problem reports no longer when a particular staff member isn't available; aliases don't have to be rearranged when people take vacations. Problems now appear in a common location accessible to all staff members. The staff person responsible for solving problems that day sees the problem report and can take the appropriate action.

Keeping track of problem reports that are not solved immediately is easier. By arranging for solved problems (appropriately marked with the "director flag" on the base note of the discussion) to be archived to another notesfile, such as "solvedproblems", only outstanding problems are kept in the notesfile. No more overflowing mailboxes, no more crowded "dothis" directories.

### Problem Logging

The "nfcomment" and "nfabort" routines allow programs to log activities and special conditions, to aid in debugging, and trivially to implement suggestion boxes.

The notesfile program makes use of both "nfcomment" and "nfabort". Notes performs a number of consistency checks on its operations. Examples include guaranteeing that the number of bytes read from a file match what was requested. When such tests fail, the program is in bad shape and it is better to stop immediately than continue. Continuing could possibly corrupt (or further corrupt) the notesfile database. "Nfabort" is used in cases like these. When notes detects internal errors, it formats a message and calls "nfabort". The message is placed in the "nfmaint" notesfile and the core image saved for

post mortem analysis. The notes administrator periodically watches the "nfmaint" notesfile and soon learns of the problem. Since all of these failures are logged it is easy to determine if the problem is an isolated one or something more serious. Problems with recreating environments are not an issue because the information is in the saved core image.

Notes uses "nfcomment" to log events such as the automatic creation of new notesfiles by the networking portions of the notes system. Problems with lock files are also logged with the nfcomment routine. (Although notes exits when a lock file is "frozen", the nfabort routine isn't used.) Nfabort generates a core image which is unnecessary for this problem.

### **Project Workbooks and Group Discussion Forums**

Notesfiles are useful for tracking project development. A notesfile (or several) can be created to hold information on design decisions, modification history, status reports, and task assignment within a project. Notes' interactive characteristics allow project members to return easily to old discussions, review design decisions, update information as needed, and mark things as completed.

A set of notesfiles provides a single, consolidated location for most information related to a project. Design decisions are not replicated in each member's mailbox; everyone can find the material when they need to. The notesfile access-list mechanism is useful for restricting the audience of the notesfile; more sensitive projects can permit only group members access to the notesfiles. Less sensitive projects may allow unlimited access, or only exclude certain users from the notesfiles.

### **Mail Processing**

Several users at UIUC use the notesfile system for all of their mail. Incoming mail is placed in a user's notesfile. Responses are properly linked with previous letters. Users use commands within the notesfile system to generate replies. Automatic copies of outgoing mail are placed in the notesfile via a switch in the user's ".mailrc" file.

Mail is kept private via restrictions in the notesfile's access list. A user may allow others to read his mail by appropriate additions to the access-list.

Non-mail generated text can also be stored in the notesfile. Users can choose to enter comments on a discussion, keeping them in the notesfile and not sending copies to other users.

### **Automatic Logs**

Periodic reports such as UUCP traffic summaries, mail backlogs, and mail routing errors are easily directed into notesfiles. Placing these reports in notesfiles allow access to the appropriate users, eliminates duplication caused by mailing to several users, and allows quick reference to previous data. If UUCP traffic is twice normal on a given day, checking to see if it is backlogged traffic from the previous day takes a single keystroke.

At UIUC, mail to the "Postmaster" mailbox is copied into a notesfile. The first staff member to scan the notesfile will perform the needed actions. Coordination of effort is much easier than if each staff member received copies of the mail; no backlogs occur when staff members are unavailable.

### **USENET**

"Notes" structures discussions better than "news". The news database doesn't maintain relationships between articles in a discussion. A new news-reading interface, "rn", provides this function but only by scanning the raw news articles to examine subject information. This is done for each session; the relationships are thrown away after the current session and not saved for the next user.

Batching is inherent in the notesfile networking mechanism. Essentially, the remote site is reading all of the new articles in one pass. This maintains the ordering of the articles and, since the presentation order is known, we can guarantee that a site won't see a base note before a response. (The exception to this is if our site sees the response before the base note and doesn't have enough information to make it an orphaned response. In this case the response becomes a base note of its own.) Orphaned responses which are later adopted by their "true parent" are properly handled. The true parent will be transmitted to the remote system during the next network transmission.

The notesfile networking programs detect network failures. Information is re-transmitted when a detectable network failure occurs. Remote UUCP failure is typically not detectable.

The Notesfile system is a general purpose discussion manager providing a single system capable of supporting problem tracking, project workbooks, mail processing, USENET-style discussions, and individual online filing systems. Notesfiles provide a common mechanism for all of these functions. The same tools can be applied to any of these applications; problems with switching between different tools are eliminated. Interacting with USENET is a small portion of the notesfile system's capabilities.